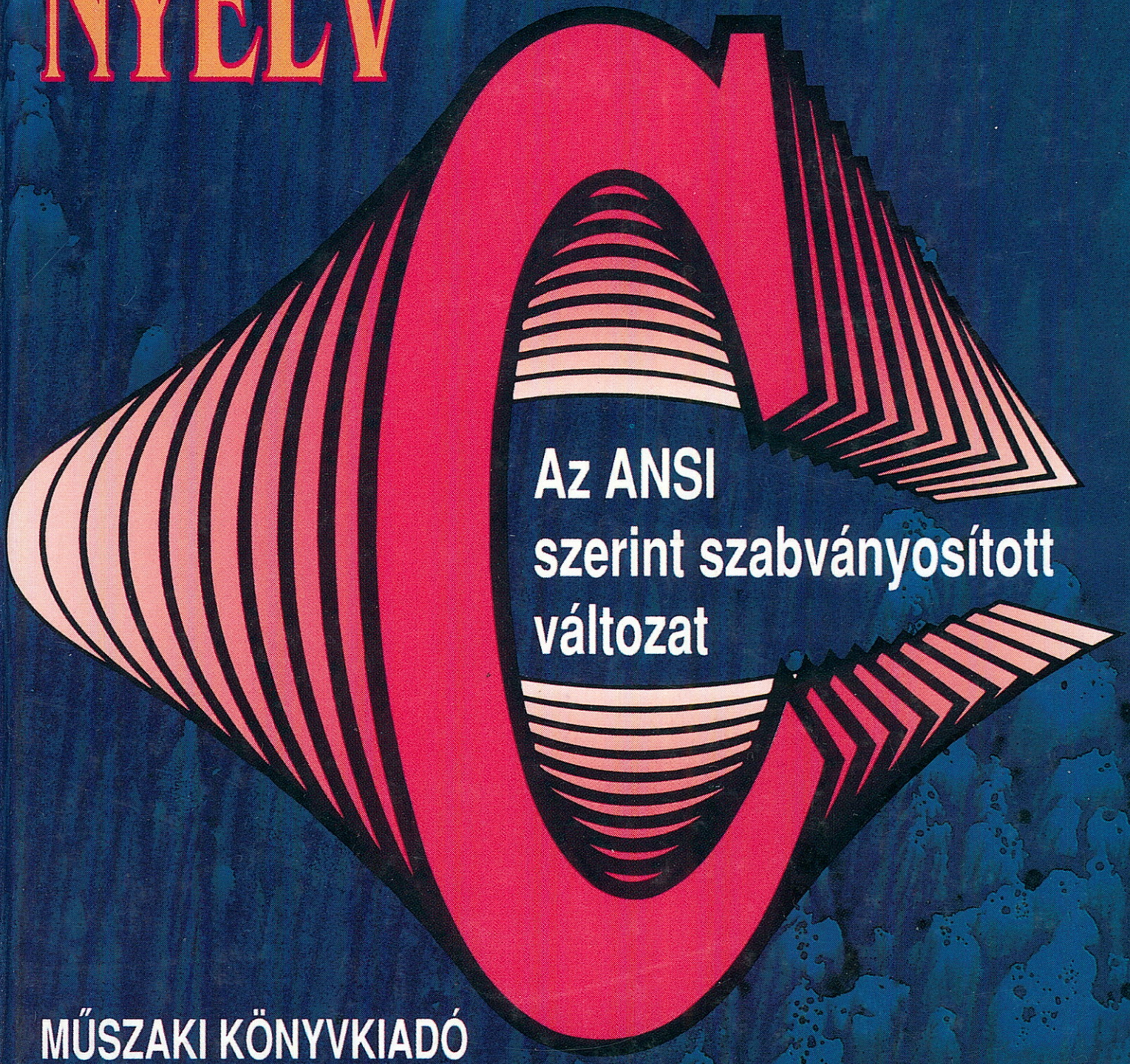
A C PROGRAMOZÁSI NYELV

Brian W. Kernighan
Dennis M. Ritchie



Az ANSI szerint szabványosított változat

MŰSZAKI KÖNYVKIADÓ

BRIAN W. KERNIGHAN—DENNIS M. RITCHIE

A C programozási nyelv

Az ANSI szerint szabványosított változat

MŰSZAKI KÖNYVKIADÓ, BUDAPEST

A könyv „A Magyar Műszaki Könyvkiadásért Alapítvány" támogatásával készült.

Az eredeti mű:
B. W. Kernighan–D. M. Ritchie
The C Programming Language
Second Edition
Original English language edition published by

Copyright © 1988, 1978 by Bell Telephone Laboratories, Incorporated
All Rights Reserved

© Hungarian translation Molnár Ervin, 1994, 2004
© Műszaki Könyvkiadó, 2004

ETO: 519.682 C
ISBN 963 16 0552 3

Kiadja a Műszaki Könyvkiadó
Felelős kiadó: Bérczi Sándor ügyvezető igazgató
Felelős szerkesztő: Kulcsár Katalin
Műszaki vezető: Abonyi Ferenc
Műszaki szerkesztő: Csoór Gabriella
A borítót tervezte: Kováts Tibor
A könyv formátuma: B/5
Ívterjedelme: 26,75 (A5)
4. kiadás
Azonosító száma: 10 220/50
A kézirat lezárva: 1996. szeptember
Készült az MSZ 5601:1983 és 5602:1983 szerint
E-mail: vevoszolg@muszakikiado.hu
Honlap: www.muszakikiado.hu

Nyomta és kötötte az Oláh Nyomdaipari Kft.
Felelős vezető: Oláh Miklós ügyvezető igazgató

Tartalom

Előszó az átdolgozott könyv magyar nyelvű kiadásához 10
Előszó az angol nyelvű második kiadáshoz 11
Előszó a könyv angol nyelvű első kiadásához 13
Bevezetés 15

1. FEJEZET: **Alapismeretek** 19
   1.1. Indulás 19
   1.2. Változók és aritmetikai kifejezések 22
   1.3. A `for` utasítás 27
   1.4. Szimbolikus állandók 28
   1.5. Karakteres adatok bevitele és kivitele 29
   1.6. Tömbök 36
   1.7. Függvények 38
   1.8. Argumentumok – az érték szerinti hívás 41
   1.9. Karaktertömbök 42
   1.10. A változók érvényességi tartománya és a külső változók 45

2. FEJEZET: **Típusok, operátorok és kifejezések** 49
   2.1. Változónevek 49
   2.2. Adattípusok és méretek 50
   2.3. Állandók 50
   2.4. Deklarációk 54
   2.5. Aritmetikai operátorok 55
   2.6. Relációs és logikai operátorok 55
   2.7. Típuskonverziók 56
   2.8. Inkrementáló és dekrementáló operátorok 60
   2.9. Bitenkénti logikai operátorok 62
   2.10. Értékadó operátorok és kifejezések 64
   2.11. Feltételes kifejezések 65
   2.12. A precedencia és a kifejezés kiértékelési sorrendje 66

3. FEJEZET: **Vezérlési szerkezetek** 69
   3.1. Utasítások és blokkok 69
   3.2. Az `if-else` utasítás 69
   3.3. Az `else-if` utasítás 71
   3.4. A `switch` utasítás 72
   3.5. Ciklusszervezés `while` és `for` utasítással 74
   3.6. Ciklusszervezés `do-while` utasítással 77
   3.7. A `break` és `continue` utasítások 78
   3.8. A `goto` utasítás és a címkék 80

4. FEJEZET: **Függvények és a program szerkezete** 82
4.1. A függvényekkel kapcsolatos alapfogalmak 82
4.2. Nem egész értékkel visszatérő függvények 86
4.3. A külső változók 88
4.4. Az érvényességi tartomány szabályai 94
4.5. A header állományok 96
4.6. A statikus változók 97
4.7. Regiszterváltozók 98
4.8. Blokkstruktúra 98
4.9. Változók inicializálása 99
4.10. Rekurzió 100
4.11. A C előfeldolgozó rendszer 102

5. FEJEZET: **Mutatók és tömbök** 107
5.1. Mutatók és címek 107
5.2. Mutatók és függvényargumentumok 109
5.3. Mutatók és tömbök 111
5.4. A címaritmetika 114
5.5. Karaktermutatók és függvények 118
5.6. Mutatótömbök és mutatókat megcímző mutatók 121
5.7. Többdimenziós tömbök 125
5.8. Mutatótömbök inicializálása 127
5.9. Mutatók és többdimenziós tömbök 127
5.10. Parancssor-argumentumok 128
5.11. Függvényeket megcímző mutatók 133
5.12. Bonyolultabb deklarációk 136

6. FEJEZET: **Struktúrák** 142
6.1. Alapfogalmak 142
6.2. Struktúrák és függvények 144
6.3. Struktúratömbök 147
6.4. Struktúrákat kijelölő mutatók 151
6.5. Önhivatkozó struktúrák 154
6.6. Keresés táblázatban 158
6.7. A `typedef` utasítás 161
6.8. Unionok 162
6.9. Bitmezők 164

7. FEJEZET: **Adatbevitel és adatkivitel** 166
7.1. A standard adatbevitel és adatkivitel 166
7.2. A formátumozott adatkivitel – a `printf` függvény 168
7.3. A változó hosszúságú argumentumlisták kezelése 170
7.4. Formátumozott adatbevitel – a `scanf` függvény 172
7.5. Hozzáférés adatállományokhoz 176
7.6. Hibakezelés – az `stderr` és `exit` függvények 179
7.7. Szövegsorok beolvasása és kiírása 180
7.8. További könyvtári függvények 182

8. FEJEZET: **Kapcsolódás a UNIX operációs rendszerhez** 186
8.1. Az állományleírók 186
8.2. Alacsony szintű adatbevitel és adatkivitel – a `read` és `write` függvények .. 187

8.3. Az `open, creat, close` és `unlink` rendszerhívások ... 189
8.4. A véletlenszerű hozzáférés – az `lseek` függvény ... 191
8.5. Példa: az `fopen` és `getc` függvények megvalósítása ... 192
8.6. Példa: katalógusok kiíratása ... 196
8.7. Példa: tárterület-lefoglaló program ... 202

A. FÜGGELÉK: **Referencia-kézikönyv** ... 207
A1. Bevezetés ... 207

A2. Lexikális megállapodások ... 207
A2.1. Szintaktikai egységek ... 207
A2.2. Megjegyzések ... 208
A2.3. Azonosítók ... 208
A2.4. Kulcsszavak ... 208
A2.5. Állandók ... 208
A2.6. Karaktersorozat-állandók ... 210

A3. A szintaxis jelölése ... 211

A4. Az azonosítók értelmezése ... 211
A4.1. A tárolási osztály ... 211
A4.2. Alapvető adattípusok ... 212
A4.3. Származtatott adattípusok ... 213
A4.4. Típusminősítők ... 213

A5. Az objektumok és a balérték ... 213

A6. Típuskonverziók ... 214
A6.1. Az egész-előléptetés ... 214
A6.2. Egészek konverziója ... 214
A6.3. Egész és lebegőpontos mennyiségek ... 214
A6.4. Lebegőpontos típusok konverziója ... 214
A6.5. Aritmetikai típuskonverziók ... 215
A6.6. Mutatók és egész mennyiségek ... 215
A6.7. A `void` típus ... 216
A6.8. A `void` típushoz tartozó mutatók ... 216

A7. Kifejezések ... 217
A7.1. Mutatógenerálás ... 217
A7.2. Elsődleges kifejezések ... 217
A7.3. Utólagos kifejezések ... 218
A7.4. Egyoperandusú operátorok ... 220
A7.5. Kényszerített típusmódosító ... 222
A7.6. Multiplikatív operátorok ... 223
A7.7. Additív operátorok ... 223
A7.8. Léptető operátorok ... 224
A7.9. Relációs operátorok ... 224
A7.10. Egyenlőségoperátorok ... 225
A7.11. Bitenkénti ÉS operátor ... 225
A7.12. Bitenkénti kizáró VAGY operátor ... 226
A7.13. Bitenkénti inkluzív VAGY operátor ... 226

A7.14. Logikai ÉS operátor ... 226
A7.15. Logikai VAGY operátor ... 226
A7.16. Feltételes operátor ... 227
A7.17. Értékadó kifejezések ... 227
A7.18. Vesszőoperátor ... 228
A7.19. Állandó kifejezések ... 228

A8. Deklarációk ... 229
A8.1. Tárolásiosztály-specifikátorok ... 230
A8.2. Típusspecifikátorok ... 230
A8.3. Struktúrák és unionok deklarációja ... 231
A8.4. Felsorolások ... 235
A8.5. Deklarátorok ... 236
A8.6. A deklarátorok jelentése ... 236
A8.7. Kezdetiérték-adás ... 240
A8.8. Típusnevek ... 242
A8.9. A `typedef` ... 243
A8.10. Típusekvivalenciák ... 243

A9. Utasítások ... 244
A9.1. Címkézett utasítások ... 244
A9.2. Kifejezésutasítások ... 244
A9.3. Összetett utasítás ... 244
A9.4. Kiválasztó utasítások ... 245
A9.5. Iterációs utasítások ... 246
A9.6. Vezérlésátadó utasítások ... 247

A10. Külső deklarációk ... 247
A10.1. Függvénydefiníciók ... 248
A10.2. Külső deklarációk ... 249

A11. Érvényességi tartomány és csatolás ... 250
A11.1. Lexikális érvényességi tartomány ... 250
A11.2. Csatolás ... 251

A12. Az előfeldolgozó rendszer ... 251
A12.1. Trigráf karaktersorozatok ... 252
A12.2. Sorok egyesítése ... 252
A12.3. Makrók definíciója és kifejtése ... 252
A12.4. Állományok beépítése ... 255
A12.5. Feltételes fordítás ... 255
A12.6. Sorvezérlés ... 257
A12.7. Hibaüzenet generálása ... 257
A12.8. A pragma direktíva ... 257
A12.9. A nulldirektíva ... 257
A12.10. Előre definiált nevek ... 257

A13. A C nyelv szintaktikájának összefoglalása ... 258

B. FÜGGELÉK: **A standard könyvtár** 266
B1. Adatbevitel és adatkivitel: az <stdio.h> header 266
B1.1. Állománykezelő műveletek 267
B1.2. Formátumozott adatkivitel 269
B1.3. Formátumozott adatbevitel 271
B1.4. Karakteres adatbevitelt és adatkivitelt kezelő függvények 273
B1.5. A közvetlen adatbevitel és adatkivitel függvényei 274
B1.6. Állományon belül pozicionáló függvények 274
B1.7. Hibakezelő függvények 275

B2. Karakteres vizsgálatok: a <ctype.h> header 275

B3. Karaktersorozat-kezelő függvények: a <string.h> header 276

B4. Matematikai függvények: a <math.h> header 278

B5. Kiegészítő rendszerfüggvények: az <stdlib.h> header 279

B6. Programdiagnosztika: az <assert.h> header 282

B7. Változó hosszúságú argumentumlisták kezelése:
az <stdarg.h> header 282

B8. Nem lokális vezérlésátadások: a <setjmp.h> header 283

B9. Jelzések kezelése: a <signal.h> header 284

B10. Dátumot és időt kezelő függvények: a <time.h> header 284

B11. A gépi megvalósításban definiált határértékek:
a <limits.h> és <float.h> headerek 286

C. FÜGGELÉK: **A C nyelv szabvány bevezetéséből adódó változásai** 289

Előszó az átdolgozott könyv magyar nyelvű kiadásához

A könyv fordítójával és szerkesztőjével együtt több más számítógépes szakember is Kernighan–Ritchie: A C programozási nyelv című könyvéből sajátította el a C nyelv alapjait. A sikert jelzi, hogy szakmai körökben a „C könyv" hivatkozás mindenki számára egyértelműen Kernighan és Ritchie könyvét jelentette, valamint az is, hogy az 1978-as angol kiadás után 1985-ben megjelent első magyar nyelvű kiadást még két további követte.

A szerzők tevőleges részvételével az ANSI a C nyelvet 1989-ben szabványosította, ami – a C nyelv filozófiáját változatlanul hagyva – számos módosítással járt. A szerzők emiatt úgy döntöttek, hogy az új szabályok és lehetőségek figyelembevételével könyvüket átírják és kibővítik. Ennek a javított kiadásnak a magyar nyelvű változatát adjuk most közre. A könyv új, javított és átdolgozott kiadása valóban jobb, teljesebb és használhatóbb a korábbinál, és mindezek alapján várható, hogy újabb programozógenerációk tankönyve és munkájuk során kézikönyve lesz.

Előszó az angol nyelvű második kiadáshoz

A számítástechnikában jelentős forradalom ment végbe a *A C programozási nyelv* című könyv első, 1978-as megjelenése óta: a nagy teljesítményű számítógépek még nagyobb teljesítményűek lettek és a személyi számítógépek teljesítőképessége megközelítette az egy évtizeddel korábbi nagy számítógépekét. Az eltelt időszakban a C programozási nyelv is megváltozott – bár csak kisebb mértékben – és már lényegesen szélesebb körben használatos, mint kezdetben, amikor a UNIX operációs rendszer nyelve volt.

A C növekvő népszerűsége, a nyelv elmúlt időszakban bekövetkezett fejlődése és a különböző C fordítók nem kellően átgondolt megvalósítása igazolja a nyelv első kiadásához képest pontosabb és korszerűbb definiálásának szükségességét. 1983-ban az Amerikai Nemzeti Szabványügyi Hivatal (ANSI) létrehozott egy bizottságot, amelynek feladata „a C nyelv egyértelmű és géptől független definiálása" volt, megtartva annak alapfilozófiáját. A munka eredménye a C nyelv ANSI szabványa lett.

A szabványos megfogalmazásra – különösen a struktúrák értékadása és kiértékelése vonatkozásában – az első kiadás már utalt, de a részletes leírást nem tartalmazta. A szabványos megfogalmazás egy új függvénydeklarálási módot tartalmaz, ami lehetővé teszi a definíciók keresztellenőrzését. Az új megfogalmazás egy standard könyvtárat is kijelöl, amelyben a be- és kivitelt kezelő függvények, a tárkezelési eljárások, karakterláncokat kezelő függvények és más, hasonló közhasznú eljárások találhatók. Az új kiadásban pontosan megadjuk az egyes nyelvi elemek viselkedését, és ezzel egy időben explicit módon megadjuk, hogy a nyelv mely vonatkozásai maradnak géptől függetlenek.

A C programozási nyelv második kiadása a C nyelvnek az ANSI szabványban definiált leírását tartalmazza. Bár megjelöltük azokat a helyeket, ahol a nyelv megváltozott, mégis a könyv megírásához kizárólag az új megfogalmazás szerinti írásmódot választottuk. A legtöbb helyen ez nem okoz számottevő eltérést és a legszembetűnőbb változást főleg a függvénydeklarációk és -definíciók mutatják. A modern C fordítóprogramok már támogatják a szabvány legtöbb jellegzetességét.

Ebben a kiadásban is megpróbáltuk megtartani az első kiadás tömörségét. A C nem egy „nagy" nyelv, nem igényel terjedelmes könyvet. Az új kiadásban igyekeztünk jobban megvilágítani a C nyelvű programozásban központi szerepet játszó kritikusabb elemeket, mint pl. a mutatókat. Az eredeti példákat szintén finomítottuk, ill. sok fejezethez új példákat is adtunk. Így például a bonyolult deklarációk kezelésével foglalkozó részt programokkal bővítettük, hogy a deklarációkat szavakba öntsük, ill. a leírást példaprogramokkal illusztráljuk. Csakúgy, mint az első kiadásban, az összes példát számítógéppel olvasható formában készítettük el és közvetlenül a szövegből kiemelve ellenőriztük.

A könyv A. Függeléke a referencia-kézikönyv, ami nem maga a szabvány, de mégis összetömörítve tartalmazza a lényeget. Ez azt jelenti, hogy a programozó számára könnyen érthető, de egy fordítóprogram írásához definícióként nem használható – ezt a szerepet csak a szabvány töltheti be megfelelően. A B. Függelék a standard könyvtár jellemzőinek

összefoglalása, és szintén csak programozói referenciaként használható, nem pedig az implementáláshoz. A C. Függelék az eredeti C változathoz képesti változásokat tartalmazza.

Mint azt az első kiadás előszavában már elmondtuk, a C nyelvet – megismerése után – egyre szívesebben használják. Egy évtizeddel több tapasztalat birtokában még mindig ugyanez a véleményünk. Reméljük, hogy ez a könyv segíteni fog a C nyelv tanulásánál és hatékony alkalmazásánál.

Mély hálával tartozunk a könyv második kiadásának elkészítését segítő barátainknak. Jon Bentley, Doug Gwyn, Doug McIlroy, Peter Nelson és Rob Pike a nyers kézirat majdnem minden oldalához értékes tanácsokat és javaslatokat adtak. Hálásak vagyunk Al Ahónak, Dennis Allisonnak, Joe Campbellnek, G. R. Emlinnek, Karen Fortgangnak, Allen Holubnak, Andrew Hume-nak, Dave Kristolnak, John Lindermannak, Dave Prossernek, Gene Spaffordnak és Chris Van Wyknek a kézirat gondos átolvasásáért. Szintén köszönetet szeretnénk mondani Bill Cheswicknek, Mark Kernighannek, Andy Koenignek, Robin Lake-nek, Tom Londonnak, Jim Reedsnek, Clovis Tondonak és Peter Weinbergernek az értékes javaslataikért. Dave Prosser számos, az ANSI szabvánnyal kapcsolatos részletkérdést válaszolt meg. Széles körben használtuk Bjarne Stroustrup C++ fordítóprogramját a példaprogramok helyi ellenőrzéséhez, és Dave Kristol bocsátotta rendelkezésünkre ANSI C fordítóprogramját a végső ellenőrzéshez. Rich Drechsler nagy segítséget nyújtott a könyv szedésében.

Őszintén köszönjük mindannyiuk áldozatos munkáját.

Brian W. Kernighan
Dennis M. Ritchie

Előszó a könyv angol nyelvű első kiadásához

A C általános célú programozási nyelv, amelyre a tömör utasításformák, a bőséges utasításkészlet és a korszerű vezérlési és adatstruktúrák jellemzőek. A C sem egy nagyon magas szintű nyelv, sem egy „nagy" nyelv, és nem egy meghatározott alkalmazási területhez készült. Mindezek ellenére a megkötések hiánya és a teljesen általános jelleg miatt a C nyelv számos, magas szintű programozási nyelvvel támogatott alkalmazási területen is kényelmesen és hatékonyan használható.

A C nyelvet eredetileg Dennis Ritchie a DEC PDP-11 UNIX operációs rendszeréhez tervezte, és a gép operációs rendszere, a C fordítóprogram, valamint az összes UNIX alkalmazói program (beleértve ezen könyv megírásához és nyomdai előkészítéséhez használt programot is) C nyelven íródott. A PDP-11-re írt változat után más gépekhez, pl. az IBM System/370-hez, a Honeywell 6000-hez és az Interdata 8/32-höz is elkészült a C fordítóprogram. Mivel a C nyelv nem kötődik egyetlen hardverhez vagy rendszerhez sem, ezért egyszerűen írhatunk olyan programokat, amelyek változtatás nélkül futtathatók bármelyik, C fordítóprogrammal ellátott gépen.

Könyvünk elsősorban azokat a tanulókat segíti, akik a C nyelvű programozást az Alapismeretek fejezet áttanulmányozása után máris megkezdhetik. A könyv további fejezetei a C nyelv főbb elemeit ismertetik, majd egy referencia-kézikönyv következik. Az egyes témakörök ismertetése elsősorban példaprogramok megértésén, írásán és módosításán alapszik, amit jobb módszernek tartunk, mint a szabályok tételes megfogalmazását. A példaprogramok többsége teljes, önálló program és nem pedig programrészlet. Az összes példát számítógéppel olvasható formában írtuk és közvetlenül a szövegből kiemelve ellenőriztük.

A nyelv hatékony használatának ismertetésén kívül – ahol lehetséges volt – igyekeztünk a stílusos, áttekinthető programozást segítő algoritmusokat és programozási elveket is bemutatni.

A könyv nem bevezető a programozástechnikába, hanem feltételezi, hogy az olvasó tisztában van olyan alapfogalmakkal, mint változó, értékadás, ciklus, függvény. Mindezek ellenére a könyvből a kezdő is elsajátíthatja a C nyelvű programozást, de esetenként szüksége lehet gyakorlottabb kollégái segítségére.

Tapasztalataink szerint a C nyelv számos alkalmazási terület programjainak kellemes, kifejező és rugalmas megfogalmazására alkalmas. Egyszerűen megtanulható, és elsajátítása után mindenki egyre szívesebben használja. Reméljük, hogy könyvünk jól fogja segíteni a C nyelv hatékony használatát.

A könyv megírása feletti örömünkhöz nagyban hozzájárultak barátaink hasznos kritikai megjegyzései és javaslatai. Különösen hálásak vagyunk Mike Bianchinak, Jim Bluenak, Stu Feldmannak, Doug McIlroynak, Bill Roome-nak, Bob Rosinnak és Larry Roslernek a kézirat több változatának gondos átolvasásáért. Köszönetet kell mondanunk Al Ahónak, Steve Bourne-nek, Dan Dvoraknak, Chuck Haley-nek, Debbi Haley-nek, Marion Harrisnek,

Rick Holtnak, Steve Johnsonnak, John Mashey-nek, Bob Mitzénak, Ralph Muhának, Peter Nelsonnak, Elliot Pinsonnak, Bill Plaugernek, Jerry Spivacknak, Ken Thompsonnak és Peter Weinbergernek a különböző munkafázisokban adott hasznos tanácsaikért. Köszönjük Mike Lesknek és Joe Ossannának a kézirat szedésénél és nyomdai előkészítésénél nyújtott segítségüket.

Brian W. Kernighan
Dennis M. Ritchie

Bevezetés

A C olyan általános célú programozási nyelv, ami szorosan kapcsolódik a UNIX operációs rendszerhez, mivel magát az operációs rendszert és a felügyelete alatt futó programok többségét is C nyelven írták. Mindezek ellenére a nyelv nem kötődik szorosan egyetlen operációs rendszerhez vagy számítógéphez sem. A C nyelvet rendszerprogramozási nyelvnek is szokás nevezni, mivel jól használható fordítóprogramok és operációs rendszerek írására, de ugyancsak hatékonyan használható különböző területek alkalmazói programjainak írásához.

A C nyelv fontosabb alapötleteinek többsége a Martin Richards által kidolgozott BCPL nyelvből ered. A BCPL C nyelvre gyakorolt hatása közvetetten, a B nyelven keresztül jelentkezik, amelyet Ken Thompson 1970-ben fejlesztett ki a DEC PDP-7 számítógépének első UNIX rendszeréhez.

A BCPL és a B nyelvek „típus nélküli" nyelvek, ellentétben a C nyelvvel, amelyben számos adattípus alkalmazható. A C nyelv alapadattípusai a karakterek, valamint a különböző méretű egész és lebegőpontos számok. Ezekhez járul a származtatott adattípusok hierarchiája, amelyekbe a mutatók, tömbök, struktúrák és unionok tartoznak. A kifejezések operátorokból és operandusokból állnak, és bármely kifejezés – beleértve az értékadást vagy a függvényhívást is – lehet önálló utasítás. A mutatókkal végzett műveletekhez a nyelv egy géptől független címaritmetikát használ.

A C nyelv tartalmazza a strukturált programozáshoz szükséges vezérlési szerkezeteket: az összetartozó utasításokat egyetlen csoportba foglaló utasítás-zárójelet, a döntési szerkezetet (`if-else`), a lehetséges esetek egyikének kiválasztását (`switch`), az elöltesztelt ciklust (`while, for`) és a hátultesztelt ciklust (`do`), valamint a ciklusból való feltétel nélküli kilépést (`break`).

A függvények értéke visszatéréskor az alapadattípusok egyike, ill. struktúra, union vagy mutató lehet. Bármely függvény rekurzívan hívható és lokális változói általában „automatikusak", vagyis a függvény minden hívásakor újra generálódnak. A függvénydefiníciók nem ágyazhatók egymásba, de a változók blokkstruktúrában is definiálhatók. Egy C program függvényei önálló forrásállományban is elhelyezhetők és külön is fordíthatók. A függvények változói belső (internal), külső, de csak egyetlen forrásállományban ismert (external) vagy a teljes programban ismert (globális) típusúak lehetnek.

A C nyelvű programok fordításához egy előfeldolgozó menet is kapcsolódik, ami lehetővé teszi a program szövegében a makrohelyettesítést (más forrásállományokat is beleértve), valamint a feltételes fordítást.

A C viszonylag alacsony szintű nyelv. Ezt a kijelentést nem pejoratív értelemben használjuk, hanem egyszerűen csak azt akarjuk kifejezni vele, hogy a C nyelv – a legtöbb számítógéphez hasonlóan – karakterekkel, számokkal és címekkel dolgozik. Ezek az alapobjektumok az adott számítógépen értelmezett aritmetikai és logikai műveletekkel kombinálhatók és mozgathatók.

A C nyelv nem tartalmaz műveleteket az összetett objektumok (karakterláncok, halmazok, listák, tömbök) közvetlen kezelésére, vagyis hiányzanak a teljes tömb vagy karakterlánc manipulálására alkalmas műveletek, bár a struktúrák egy egységenkénti

másolása megengedett. A nyelvben csak a statikus és a függvények lokális változóihoz használt verem típusú tárfoglalási lehetőség létezik, és nincs a más nyelvekben megszokott heap vagy garbage collection (a felszabaduló tárterületeket összegyűjtő és hasznosító mechanizmus) típusú dinamikus tárkezelés. Végül pedig a C nyelvben nincs adatbeviteli és adatkiviteli lehetőség, azaz nincs READ vagy WRITE utasítás, valamint nincsenek beépített állományelérési módszerek sem. Mindezeket a magasabb szintű tevékenységeket explicit függvényhívásokkal kell megvalósítani. A legtöbb C implementáció szerencsére már tartalmazza ezen tevékenységek megfelelő gyűjteményét, az ún. standard könyvtárat.

További jellemzője a C nyelvnek, hogy csak egy tevékenységi sorrendnek megfelelő vezérlő szerkezeteket – ellenőrzés, ciklus, utasításcsoport, alprogram – tartalmaz és nem teszi lehetővé a multiprogramozást, a párhuzamos műveletvégzést, a folyamatok szinkronizálását vagy a korutinok (párhuzamos rutinok) alkalmazását.

Bár ezen lehetőségek némelyikének hiánya komoly hiányosságnak tűnik („Két karakterlánc összehasonlításához egy függvény szükséges?"), a nyelv szigorú korlátozása valójában előnyös. Mivel a C viszonylag „kis" nyelv, ezért tömören leírható és gyorsan megtanulható. A programozótól elvárható, hogy ismerje és értse, valamint szabályosan használja a teljes nyelvet.

Éveken keresztül *A C programozási nyelv* első kiadásában szereplő referencia-kézikönyv volt a C nyelv definíciója. 1983-ban az Amerikai Nemzeti Szabványügyi Intézet (ANSI) létrehozott egy bizottságot a C nyelv modern, átfogó definiálására. Az így kapott definíció a C ANSI szabványa vagy röviden az ANSI C, amely 1988-ban vált teljessé. A modern fordítóprogramok ma már a szabvány előírásainak többségét támogatják.

A szabvány az eredeti referencia-kézikönyvön alapszik. A nyelv viszonylag keveset változott, mivel a szabvány megalkotásakor az egyik célkitűzés az volt, hogy a már meglévő programok többsége változatlanul használható legyen vagy ennek hiányában a fordítóprogram legalább figyelmeztessen a változásra.

A legtöbb programozó számára a legfontosabb eltérést a függvények deklarálásának és definiálásának megváltozott szintaktikája jelenti. A függvénydeklaráció új változata magában foglalja a függvény argumentumainak leírását. Ez a járulékos információ megkönnyíti a fordítóprogram számára az argumentumok hibás illesztéséből adódó hibák detektálását. Tapasztalataink szerint ez hasznos bővítése volt a nyelvnek.

Van néhány további, kisebb változás is a nyelvben: a struktúrák értékadása és kiértékelése, amelyet széles körben használtak, most a nyelv hivatalos részévé vált. A lebegőpontos számítások egyszeres (single) pontossággal is elvégezhetők. Az aritmetika tulajdonságait, különösen az előjel nélküli adattípusok esetén, tisztázta az új szabvány. Az előfeldolgozó (preprocesszor) rendszer sokkal kimunkáltabb lett. Ezen változások zöme csak kis mértékben érinti a legtöbb programozót.

A szabvány másik jelentős vonatkozása a C könyvtár kialakítása. Ez olyan függvényeket tartalmaz, amelyek többek között lehetővé teszik az operációs rendszerhez való hozzáférést (pl. állományok olvasása és írása), a formátumozott adatbevitelt és adatkivitelt, a tárkiosztás szervezését, a karakterláncokkal végzett műveleteket. Az ún. szabványos fejek (headerek) gyűjteménye lehetővé teszi a függvény- és adattípus-deklarációk egységes kezelését. Ezt a könyvtárat használó programok kompatíbilis módon fognak együttműködni a befogadó rendszerrel. A könyvtár jelentős része a UNIX rendszer standard I/O könyvtárát modellezi. Ezt a könyvtárat a könyv első kiadásában már leírtuk, és széles körben használták más rendszerekhez is. A legtöbb programozó ebben sem talál sok változást.

Mivel a C nyelvben alkalmazott adattípusok és vezérlési szerkezetek alkalmazását a legtöbb számítógép közvetlenül támogatja, az önmagában zárt programok formájában megvalósított futtatási könyvtár kicsi. A standard könyvtár függvényeit csak

explicit módon hívjuk, így minden további nélkül elhagyhatók, ha nincs szükség rájuk. A függvények többsége C nyelven íródott és – az operációs rendszerhez tartozó részek kivételével – más gépre is átvihető.

A C nyelv sokféle számítógép adottságaihoz illeszkedik, mégis bármilyen konkrét számítógép felépítésétől független, ezért viszonylag kis fáradsággal írhatunk hordozható, azaz változtatás nélkül különféle számítógépeken futtatható, C programokat. A szabvány a hordozhatóságot explicit módon megköveteli, és azon számítógép jellemzésére, amelyen a program futtatható egy paraméterhalmazt ír elő.

A C nem nevezhető erősen típusos nyelvnek, de a fejlődése során a típusellenőrzés erősödött. A C eredeti definíciója, eléggé el nem ítélhető módon, megengedte a mutatók és az egész típusú adatok keverését. Ezt a hiányosságot már régen kiküszöbölték, és a szabvány már megköveteli a megfelelő deklarációt és az explicit típuskonverziót, amit a jó fordítóprogramok ki is kényszerítenek. A függvénydeklaráció új formája a másik olyan lépés, ami a típusellenőrzés szigorodása irányába mutat. A fordítóprogramok a legtöbb típusillesztési hibára figyelmeztetnek és inkompatíbilis adatok között nincs automatikus típuskonverzió. Bárhogyan is nézzük, a C megtartotta az alapfilozófiáját, miszerint a programozónak csak tudnia kell, hogy mit csinál, és a C nyelv csak azt igényli, hogy a szándékát egyértelműen fogalmazza meg.

A C, hasonlóan más nyelvekhez, nem hibátlan. Némelyik művelet rossz precedencia szerint megy végbe és a szintaxis néhány helyen jobb is lehetne. Mindezek ellenére a C különböző programozási területeken rendkívül hatásos és kifejező nyelvnek bizonyult.

Végezetül még néhány szót szeretnénk szólni a könyv felépítéséről: az 1. fejezet a C nyelv főbb részeinek áttekintése, aminek az a célja, hogy az olvasó a lehető leghamarabb elkezdhesse a programok írását. Véleményünk szerint egy új nyelv megtanulásának legjobb módja, ha az adott nyelven programokat írunk. Az 1. fejezet feltételezi, hogy az olvasó rendelkezik az alapvető programozástechnikai ismeretekkel, ezért nem foglalkozunk azzal, hogy mi a számítógép vagy mi a fordítás, és nem magyarázzuk pl. az `n = n+1` típusú kifejezések értelmezését sem. Ahol lehetőség volt rá, megpróbáltunk hasznos programozási módszereket bemutatni, de a könyvet nem az adatstruktúrák és algoritmusok kézikönyvének szántuk, így ahol kénytelenek voltunk választani, inkább a nyelv leírására helyeztük a hangsúlyt.

A 2.-tól a 6. fejezetig terjedő részben az 1. fejezetben leírtaknál részletesebben és precízebben mutatjuk be a C nyelv egyes elemeit. A hangsúly itt is a teljes példaprogramokon van, az egyes elemeket illusztráló részletek helyett. A 2. fejezet az alapvető adattípusokkal, operátorokkal és kifejezésekkel foglalkozik. A 3. fejezet a vezérlési szerkezeteket (`if-else, switch, while, for` stb). tekinti át. A 4. fejezet témája a függvények és a program szerkezete, a külső változókkal és az érvényességi tartománnyal kapcsolatos problémák, valamint a több forrásállományú feldolgozás kérdései. Érintőlegesen itt tárgyaljuk az előfeldolgozó (preprocesszor) rendszert is. Az 5. fejezet a mutatókkal és a címaritmetikával, a 6. fejezet pedig a struktúrákkal és unionokkal foglalkozik.

A 7. fejezet témája az operációs rendszer felé közös csatlakozási felületet adó standard könyvtár. Ezt a könyvtárat az ANSI szabvány definiálja és minden C nyelv használatát lehetővé tevő számítógép támogatja, ezért az adatbevitelt és adatkivitelt, ill. más operációsrendszer-hívásokat tartalmazó programok változtatás nélkül átvihetők az egyik rendszerről a másikra.

A 8. fejezet a C nyelvű programok és a UNIX operációs rendszer közti kapcsolatot írja le, a hangsúlyt az adatbevitelre és -kivitelre, az állománykezelésre és a tárkiosztásra helyezve. A fejezet néhány része UNIX-specifikus, de más operációs rendszer alatt dolgozó programozók is haszonnal olvashatják, mivel megtudható belőle, hogy hogyan alakítható ki a standard könyvtár adott változata vagy hogyan érhető el a programok hordozhatósága.

Az A. Függelék a nyelv referencia-kézikönyve, és mint ilyen, tartalmazza a C nyelv szintaktikájának és szemantikájának ANSI szabvány szerinti hivatalos leírását. Ez a rész elsősorban a C fordítóprogramok írásához nyújt segítséget. A referencia-kézikönyv a nyelv definícióját nagyon tömör formában adja meg. A B. Függelék a standard könyvtárra vonatkozó ismeretek összefoglalása és szintén inkább a felhasználóknak, mint a könyvtárat megvalósítani akaróknak szól. A C. Függelék az ANSI szabvány eredeti nyelvtől való eltéréseit foglalja össze. Kétséges esetekben a szabványt és a saját fordítóprogramunkat tekintettük mérvadónak.

Alapismeretek

Kezdjük a C nyelv tanulását az alapfogalmakkal! Az a célunk, hogy a nyelv elemeit működőképes programokon keresztül mutassuk be, anélkül, hogy belemennénk a részletekbe, formális szabályokba és a kivételek tárgyalásába. Ezért nem törekszünk a teljességre vagy pontosságra, de természetesen ettől függetlenül a leírt példák helyesek. El szeretnénk érni, hogy az olvasó a lehető leggyorsabban hasznos kis programokat írjon, emiatt ebben a fejezetben csak az alapfogalmakra (változók, állandók, aritmetika, vezérlési szerkezetek, függvények, ill. az egyszerű adatbevitel és -kivitel) koncentrálunk. Szándékosan nem foglalkozunk a C nyelv olyan lehetőségeivel, amelyek elsősorban a nagyobb programok írásánál szükségesek. Ezek közé tartozik a mutatók és struktúrák használata, a C nyelv gazdag operátorkészletének jelentős része, néhány vezérlési szerkezet és a standard könyvtár.

Ennek a megközelítésnek természetesen hátrányai is vannak: a legsúlyosabb, hogy a nyelv egy elemét leíró összes információt a fejezet rövidsége miatt itt nem adhatjuk meg és ebből félreértések keletkezhetnek. A másik gond, hogy a példaprogramok nem használhatják ki a C nyelv összes lehetőségét, így nem olyan tömörek és elegánsak, mint ahogy szeretnénk. Mindent elkövettünk, hogy ezeket a hátrányokat csökkentsük, de kérjük az olvasót, hogy az itt elmondottakat vegye figyelembe a fejezet tanulmányozása során. Az előzőek miatt a későbbi fejezetekben kénytelenek leszünk ismétlésekbe bocsátkozni, de reméljük, hogy ez inkább segíti az olvasót a megértésben, mintsem bosszantaná.

A tapasztalt programozók természetesen már ebből a fejezetből is kikövetkeztethetik a számukra szükséges további tudnivalókat. A kezdőknek javasoljuk, hogy az itteni példákhoz hasonló kis programokat írjanak.

Az 1. fejezetet a kezdő és tapasztalt programozók egyaránt keretként használhatják a 2. fejezettel kezdődő részletes leíráshoz.

1.1. Indulás

Egy új programozási nyelv elsajátításának egyetlen útja, hogy az adott nyelven programokat írunk. Az első példaprogram minden nyelv tanulásának kezdetén előfordul. A feladat, hogy *nyomtassuk ki a következő szöveget:*

```
Halló mindenki!
```

A feladat megoldása számos problémát vet fel: képesnek kell lennünk egy program létrehozására, annak sikeres lefordítására, betöltésére, futtatására, és ki kell találnunk, hogy a kiírt szöveg hol jelenik meg. Ezeken a rutin jellegű részleteken túljutva a többi már viszonylag egyszerű.

A C nyelvben a „Halló mindenki!" szöveget kiíró program a következő módon néz ki:

```
#include <stdio.h>
main ( )
{
   printf ("Halló mindenki!\n");
}
```

A program futtatásának módja az általunk használt rendszertől függ. Például a UNIX operációs rendszer alatt a programot egy olyan forrásállományban kell létrehozni, amelynek neve `.c`-re végződik. Ilyen név lehet pl. az, hogy `hallo.c`. Az így elkészített forrásprogramot a

```
cc hallo.c
```

paranccsal le kell fordítani.

Ha a program beírásakor nem hibáztunk (pl. nem hagytunk ki betűket vagy nem írtunk valamit hibásan), akkor a fordítás rendben megtörténik és egy `a.out` nevű végrehajtható állomány keletkezik. Ezt a

```
a.out
```

paranccsal futtathatjuk. Ennek hatására a kimeneten megjelenik a

```
Halló mindenki!
```

szöveg.

Más operációs rendszer alatt dolgozva természetesen más szabályok érvényesek. Ha gondunk támad, forduljunk a helyi viszonyokat ismerő szakemberhez.

Most némi magyarázatot fűzünk a mintaprogramunkhoz. Egy C nyelvű program, bármilyen méretű is legyen, *függvényekből* és *változókból* áll. A függvény utasításokat tartalmaz, amelyek meghatározzák, hogy a számítás menetét hogyan kell végrehajtani, és a változók azokat az értékeket tárolják, amelyekkel a számolást végre kell hajtani. A C nyelv függvényei hasonlóak a FORTRAN szubrutinjaihoz, ill. függvényeihez vagy a Pascal eljárásaihoz, ill. függvényeihez. A példaprogramunk egy `main` nevű függvényből áll. Általában a függvény neve tetszőleges lehet, de a `main` egy speciális név és a program végrehajtása mindig a `main` elején kezdődik. Ebből következik, hogy minden programban valahol kell hogy legyen egy `main`.

A `main` a feladat végrehajtása érdekében általában további függvényeket hív, és ezeket vagy a programmal együtt mi írjuk, vagy a függvénykönyvtárban találhatók. A példaprogram első

```
#include <stdio.h>
```

sora éppen azt mondja meg a fordítóprogramnak, hogy a fordítás során a programba foglalja bele a standard bemeneti/kimeneti könyvtárra vonatkozó információkat. Ez a sor a legtöbb C nyelvű forrásprogram elején megtalálható. A standard könyvtárat a 7. fejezetben és a B. Függelékben írjuk le.

A függvények közti adatcsere egyik módszere, hogy a hívó függvény adatokból álló listát, az ún. *argumentumokat* adja át a hívott függvénynek. A függvény neve utáni ( ) zárójelek ezt az argumentumlistát határolják. A mintapéldánkban a `main`-t olyan függvényként definiáltuk, amelynek nincs argumentuma és ezt a ( ) üres lista jelöli.

```
#include <stdio.h>
```
beépíti a standard könyvtárra vonatkozó információkat;

```
main( )
```
definiál egy függvényt `main` *névvel, argumentumok nélkül;*

```
{
```
kapcsos zárójelek határolják

```
printf("Halló mindenki!\n");
```
a `main`*-t alkotó utasításokat;* `main` *a* `printf` *könyvtári függvényt hívja a kívánt szöveg kiíratásához; a* `\n` *egy újsor-karaktert jelöl*

```
}
```

Az első C nyelvű program

A függvényt alkotó utasításokat a { } kapcsos zárójelek határolják. A `main` függvény csak egyetlen utasítást tartalmaz:

```
printf("Halló mindenki!\n");
```

Egy függvényt a nevével és az azt követő, zárójelben elhelyezett argumentumlistával hívunk, így ez az utasítás a `printf` függvény hívása a `"Halló mindenki!\n"` argumentummal. A `printf` egy köyvtári függvény, amely az idézőjelek közti szöveget *(karakterláncot)* a kimeneti egységre írja ki.

A két felső idézőjel közt elhelyezkedő tetszőleges számú karakterből álló szöveget *karakterláncnak (stringnek)* vagy *karakterlánc-állandónak* (stringkonstansnak) nevezzük. Ilyen karakterlánc a mi esetünkben a `"Halló mindenki!\n"`. A könyv ezen bevezető részében a karakterláncokat csak a `printf` és más függvények argumentumaként használjuk.

A karakterláncban lévő `\n` jelsorozat a C nyelvben az újsor-karaktert jelöli, amelynek kiírása után a további szöveg a következő sor elején (bal margójánál) kezdve következik. Ha a `\n` jelsorozatot elhagyjuk (javasoljuk ennek kipróbálását), akkor a kiírás után a kocsi vissza és a soremelés kiírása elmarad. A `printf` függvény argumentumában az újsor-karaktert csakis a `\n` jelsorozat beiktatásával kérhetjük, és ha megpróbálkoznánk az argumentum

```
printf("Halló mindenki!
");
```

típusú beírásával, akkor a C fordítóprogram hibát jelezne.

A `printf` soha nem hoz létre automatikusan soremelést, így egyetlen sornyi szöveg `printf` hívás sorozataként is kiíratható. Ennek megfelelően az első példaprogramunkat így is írhattuk volna:

```
#include <stdio.h>

main( )
{
        printf("Halló ");
        printf("mindenki!");
        printf("\n");
}
```

Mindkét változat azonos módon nyomtatja ki a kívánt szöveget.

Megjegyezzük, hogy a `\n` csak egyetlen karaktert jelent. A `\n` jelsorozathoz hasonló, ún. *escape jelsorozatok* általános és jól bővíthető lehetőséget nyújtanak a nehezen nyomtatható vagy nyomtatási képpel nem rendelkező karakterek előállítására. A C nyelvben ilyen escape jelsorozat még a `\t` a tabulátor, `\b` a kocsi-visszaléptetés (backspace), `\"` az idézőjel vagy `\\` a fordított törtvonal (backslash) előállítása. Az escape jelsorozatok teljes listája a 2.3. pontban található.

1.1. gyakorlat. Futtassa le a `"Halló mindenki!"` szöveget kiíró programot a saját rendszerén! Próbálja meg a program egyes részeit elhagyni és figyelje meg milyen hibajelzést ad a rendszer!

1.2. gyakorlat. Próbálja ki, hogy mi történik, ha a `printf` argumentumában a `\x` jelsorozat szerepel (ahol `\x` a fenti listában nem szereplő escape jelsorozat)!

1.2. Változók és aritmetikai kifejezések

A következő példaprogrammal a °C = (5/9) (°F–32) képlet alapján, táblázatos formában ki akarjuk íratni az összetartozó, Fahrenheit- (°F) és Celsius-fokban (°C) mért hőmérsékletértékeket a következő formában:

```
0      -17
20     -6
40      4
60     15
80     26
100    37
120    48
140    60
160    71
180    82
200    93
220    104
240    115
260    126
280    137
300    148
```

A program most is egyetlen, `main` nevű függvény definiálásából áll, hosszabb az előző példaprogramnál, de nem bonyolultabb annál. Ebben a programban már néhány új fogalmat (megjegyzés, deklaráció, változók, aritmetikai kifejezések, ciklus, formátumozott adatkivitel) is bevezetünk. Maga a példaprogram a következő:

```
#include <stdio.h>

/* Fahrenheit-fok–Celsius-fok táblázat kiírása
F = 0, 20, ..., 300 Fahrenheit-fokra */
main( )
{
        int fahr, celsius;
        int also, felso, lepes;
```

```
        also = 0;     /* a táblázat alsó határa */
        felso = 300; /* a táblázat felső határa */
        lepes = 20;  /* a táblázat lépésköze */

        fahr = also;
        while (fahr <= felso) {
            celsius = 5 * (fahr-32) / 9;
            printf("%d\t%d\n", fahr, celsius);
            fahr = fahr + lepes;
        }
}
```

A program első két sora, a

```
/* Fahrenheit-fok–Celsius-fok táblázat kiírása
F = 0, 20, ..., 300 Fahrenheit-fokra */
```

sorok egy *megjegyzést* alkotnak *(comment),* amely röviden leírja a program működését és feladatát. Bármilyen, `/*` és `*/` közt elhelyezkedő szöveget a C fordítóprogram figyelmen kívül hagy, ezért ide tetszőleges, a program megértését és használatát segítő szöveget írhatunk. A programban bárhol lehet megjegyzés, ahol betűköz, tabulátor vagy új sor előfordulhat.

A C nyelvben minden változót a használata előtt deklarálni kell, ami általában a függvény kezdetén, a végrehajtható utasítások előtt történik. A *deklaráció* a változók tulajdonságait írja le és egy típus megadásából, valamint az adott típusú változók felsorolásából áll, mint pl.

```
int fahr, celsius;
int also, felso, lepes;
```

Az `int` típus azt jelenti, hogy a felsorolt változók egész (integer) értéket vehetnek fel, ellentétben a `float` típus megadásával, amely lebegőpontos értékű változót – azaz olyan változót, amelynek értéke törtrészt is tartalmaz – jelöl. Az `int` és `float` típusú változók pontossága és lehetséges nagysága a használt számítógéptől függ. Gyakran 16 bites `int` típusú változókat használnak, amelyek értéke –32 768 és +32 767 közé eshet, de előfordul 32 bites `int` típusú változó is. A `float` típusú számokat általában 32 biten ábrázolják, legalább hat értékes számjegy pontossággal és az abszolút értékük 10^{-38}-tól 10^{+38}-ig terjedhet.

Az `int` és `float` típuson kívül a C nyelv még további adattípusokat is értelmez. Ilyen a

`char` karakter, egy bájton ábrázolva;
`short` rövid egész típusú szám;
`long` hosszú egész típusú szám;
`double` kétszeres pontosságú lebegőpontos (valós) szám.

Ezen adattípusok méretei szintén a felhasznált számítógéptől függenek. Ezekből az elemi (alap)adattípusokból épülnek fel az összetett adattípusok: *tömbök, struktúrák* és *unionok.* Az adott típusú adatokra *mutatók (pointerek)* mutathatnak és a *függvények* adott típusú értékkel térnek vissza az őket hívó függvényhez. Mindezekről a későbbiekben még szó lesz. A hőmérséklet-átszámító program számítási műveletei az

```
also = 0;
felso = 300;
lepes = 20;
fahr = also;
```

értékadó utasításokkal kezdődnek, amelyek a felhasznált változók kezdeti értékeit állítják be. Az egyes utasításokat a pontosvessző zárja.

A táblázat minden sorát azonos módon kell kiszámítani, így a soronként egyszer ismétlődő számítások megvalósítására ciklust használunk, amelyet a `while` utasítással alakítunk ki a következő módon:

```
while (fahr <= felso) {
    ...
}
```

A `while` utasítással szervezett ciklus működése a következő: futás közben a számítógép megvizsgálja a zárójelben elhelyezett kifejezést, és ha az igaz (`fahr` kisebb vagy egyenlő, mint `felso`), akkor végrehajtja a kapcsos zárójelek közti ciklusmagot (az esetünkben ez három utasításból áll). Ezután a gép a feltételt újra megvizsgálja, és ha ismét igaz az értéke, akkor a ciklusmag újra végrehajtódik. Ha egyszer a vizsgálat eredménye hamis lesz (`fahr` értéke nagyobb lesz, mint `felso`), a ciklus befejeződik, és a program a ciklust követő utasítással folytatódik. A példaprogramunkban a ciklus után már nincs újabb utasítás, így a program befejeződik.

A `while` utasítás ciklusmagja egy vagy több, kapcsos zárójelek közt elhelyezett utasításból (mint a hőmérséklet-átalakító programban) vagy egyetlen, kapcsos zárójelek nélkül elhelyezett utasításból állhat. Ez utóbbit szemlélteti a következő példa:

```
while (i < j)
    i = 2 * i;
```

A `while` hatáskörébe tartozó utasításokat mindkét esetben beljebb írtuk, hogy világosan kitűnjön, mely utasítások tartoznak a ciklusmaghoz. Ez a beljebb kezdés a program logikai szerkezetét hangsúlyozza. A C fordítóprogramok eléggé kötetlenül kezelik az utasítások elhelyezését, a bekezdések és üres helyek csak a program olvasását és megértését segítik. Célszerű, ha soronként csak egy utasítást írunk és az operátorok előtt, ill. után írt szóközzel tesszük egyértelművé a tagolást. A zárójelek elhelyezkedése kevésbé kritikus, erre egyéni stílust alakíthatunk ki, vagy átvehetjük valamelyik, éppen divatos stílust. Bármilyen, nekünk tetsző formát választhatunk, de célszerű, ha a későbbiekben ehhez következetesen ragaszkodunk.

A program a munka zömét a ciklusmagban végzi. A Celsius-fokban mért hőmérsékletet kiszámító és a Celsius nevű változónak értékül adó utasítás

```
celsius = 5 * (fahr-32) / 9;
```

Az ok, ami miatt először 5-tel szorzunk, majd 9-cel osztunk az 5/9-del való szorzás helyett az, hogy a C nyelv – több más programozási nyelvhez hasonlóan – az egész számok osztásánál csonkít, az eredmény törtrészét elhagyja. Mivel 5 és 9 egész számok, 5/9-ed csonkított eredménye nulla, így minden Celsius-fok értéke nulla lenne.

Ez a példaprogram egy kicsivel többet is bemutat a `printf` függvény működéséből. A `printf` általános célú, formátumozott kimenetet előállító függvény, amelyet részletesen a 7. fejezetben fogunk ismertetni. A függvény első argumentuma a ki-

nyomtatandó karakterlánc, amelyben a % jelek mutatják, hova kell az első (második, harmadik stb.) argumentum(ok) értékét behelyettesíteni és milyen formában kell azokat kiírni. Például a %d egy egész típusú argumentumot jelöl ki, így a

```
printf("%d\t%d\n", fahr, celsius);
```

utasítás két egész típusú változó (`fahr` és `celsius`) értékét fogja kiírni, köztük egy tabulátort (`\t`) elhelyezve. Az első argumentum minden egyes % jeles konstrukciójához egy megfelelő második, harmadik stb. argumentum párosul. A % konstrukciókból és a további argumentumokból álló párroknak szám és típus szerint meg kell egyeznie, különben hibás eredményt kapunk.

Egyébként a `printf` nem része a C nyelvnek, a nyelvben magában nincs definiálva az adatbevitel és -kivitel. A `printf` csak egy hasznos függvény, ami a C programok által hozzáférhető standard könyvtárban található. A `printf` viselkedését az ANSI szabvány definiálja, így a szabványon keresztül a függvény tulajdonságai minden fordítóprogram és könyvtár számára azonosak.

Azért, hogy a figyelmünket főleg a C nyelvnek szentelhessük, az adatok beviteléről és kiviteléről a 7. fejezetig nem sokat beszélünk. Elsősorban a formátumozott adatátvitel tárgyalását halasztjuk későbbre. Ha numerikus adatokat akarunk a programmal beolvastatni, akkor a 7.4. pontban olvassuk el a `scanf` függvényre vonatkozó részeket. A `scanf` hasonló a `printf` függvényhez, csak adatkiírás helyett adatot olvas.

A hőmérséklet-átalakító programunknak számos baja van. Az egyik legegyszerűbben megszüntethető hiba, hogy a kiírás nem túl szép, mivel a számok nincsenek jobbra igazítva. Ezen könnyű segíteni: ha a `printf` utasításban lévő `%d` konstrukciót a szélességet megadó résszel egészítjük ki, akkor a számok a rendelkezésükre álló mezőben jobbra igazodva jelennek meg. Például azt írhatjuk, hogy

```
printf("%3d %6d\n", fahr, celsius);
```

akkor az egyes sorokban az első szám három számjegy széles, a második szám pedig hat számjegy széles mezőbe íródik az alábbiak szerint:

```
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
...    ...
```

A legkomolyabb probléma az egész aritmetika használatából adódik, mivel a kapott Celsius-fok értékek nem túl pontosak. Pl. a 0 °F-nek a –17,8 °C felel meg és nem pedig a táblázatban szereplő –17 °C. Pontosabb eredményt kapunk, ha az egészaritmetika helyett lebegőpontos aritmetikát használunk. Ez a program kis változtatását igényli. Ennek megfelelően a program második változata:

```
#include <stdio.h>

/* Fahrenheit-fok-Celsius-fok táblázat kiírása F = 0, 20,
..., 300 Fahrenheit-fokra; a program lebegőpontos változa-
ta */
```

```
main( )
{
    float fahr, celsius;
    int also, felso, lepes;

    also = 0;       /* a táblázat alsó határa */
    felso = 300;    /* a táblázat felső határa */
    lepes = 20;     /* a táblázat lépésköze */

    fahr = also;
    while (fahr <= felso) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + lepes;
    }
}
```

A program lényegében azonos az előző változattal, csak a `fahr` és `celsius` változók `float`-ként lettek deklarálva és az átalakítást megadó képletet sokkal inkább a megszokott formában írtuk. Az előző programban nem írhattunk `5/9`-det, mert az egészosztás okozta csonkítás miatt az eredmény nulla lett volna. Az új képlet állandóiban a tizedespont jelzi, hogy lebegőpontos számokról van szó, így az `5.0/9.0` nem csonkul, hiszen két lebegőpontos szám hányadosa is lebegőpontos szám.

Ha egy aritmetikai operátornak egész típusú operandusai vannak, akkor a gép az egész számokra érvényes műveletet fogja elvégezni. Ha egy aritmetikai operátor egyik operandusa lebegőpontos, a másik pedig egész típusú, a művelet végrehajtása előtt az egész típusú operandus automatikusan lebegőpontossá konvertálódik. Amennyiben a képletben `fahr-32` szerepelne, a `32` automatikusan lebegőpontos számmá alakulna. Ennek ellenére célszerű a lebegőpontos állandókban a tizedespontot akkor is kiírni, ha a szám éppen egész értékű, mivel ez az olvasó számára jobban kihangsúlyozza a szám lebegőpontos jellegét.

Az egész típusú adatok lebegőpontossá alakításának részletes szabályaival a 2. fejezetben foglalkozunk. Pillanatnyilag csak azt jegyezzük meg, hogy a

```
fahr = also;
```

értékadás, valamint a

```
while (fahr <= felso)
```

vizsgálat az előbb elmondottak szerint működik, azaz az `int` típusú adatok a végrehajtás előtt `float` típusúvá alakulnak.

A `printf` függvényben szereplő `%3.0f` konverziós előírás azt jelenti, hogy a lebegőpontos szám (a mi esetünkben a `fahr`) legalább három karakter széles mezőbe lesz kinyomtatva, tizedespont és törtrész nélkül. A `%6.1f` egy másik szám (a `celsius`) kiírását specifikálja: ez legalább hat karakter széles mezőben lesz kinyomtatva, amiből egy számjegy a tizedespont után van. Az így kiírt táblázat a következő:

```
  0     -17.8
 20      -6.7
 40       4.4
...       ...
```

A szélesség vagy pontosság hiányozhat is a specifikációból: a `%6f` azt írja elő, hogy a

szám legalább hat karakter széles mezőbe nyomtatódik; a `%.2f` azt, hogy a számnak a tizedespont után még két karaktere lehet és a teljes szélességére nincs előírás; a `%f` pedig pusztán csak azt jelzi, hogy a számot lebegőpontos formában kell kiírni. A következőben bemutatunk néhány formátumspecifikációt:

`%d`	a számot decimális egészként írja ki;
`%6d`	a számot decimális egészként, legalább hat karakter széles mezőbe írja ki;
`%f`	a számot lebegőpontosként írja ki;
`%6f`	a számot lebegőpontosként, legalább 6 karakter széles mezőbe írja ki;
`%.2f`	a számot lebegőpontosként, két tizedessel írja ki;
`%6.2f`	a számot lebegőpontosként, legalább 6 karakter széles mezőbe, két tizedessel írja ki.

Többek közt a `printf` függvény a `%o` specifikációt az oktális, a `%x` specifikációt a hexadecimális, a `%s` specifikációt a karakterlánc típusú kiíráshoz, és a `%%` specifikációt pedig a `%` jel kiírására használja.

1.3. gyakorlat. Módosítsuk a hőmérséklet-átalakító programot úgy, hogy a táblázat fölé fejlécet is nyomtasson!

1.4. gyakorlat. Írjunk programot, amely a Celsius-fokban adott értékeket alakítja Fahrenheit-fokká!

1.3. A for utasítás

Az egyes feladatok megoldására többféle módon írhatunk programot. Próbáljuk meg a hőmérséklet-átalakító programunk következő változatát:

```
#include <stdio.h>

/* Fahrenheit-fok-Celsius-fok átszámítási táblázat */
main( )
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr+20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Ez a program ugyanazt csinálja, mint az előző, de attól szemlátomást különbözik. Az egyik legjelentősebb változás, hogy eltűnt a változók többségének deklarálása, csak a `fahr` maradt meg, `int` típusúként. Az alsó és felső határt, ill. a lépésközt csak állandóként szerepeltetjük a `for` utasításban, ami maga is új a számunkra. A Celsius-fokot kiszámító kifejezés sem önálló utasítás; hanem a `printf` függvény harmadik argumentumaként szerepel.

Ez az utóbbi változtatás egy teljesen általános szabályra mutat példát: minden olyan összefüggésben, ahol valamilyen típusú változó értékét használjuk, megengedett egy ugyanolyan típusú összetett kifejezés használata is. Mivel a `printf` harmadik argumentumának a `%6.1f` specifikációhoz illeszkedően egy lebegőpontos számnak kell lennie, ezen a helyen bármilyen lebegőpontos kifejezés megadható.

A `for` utasítás szintén egy ciklusszervező utasítás, a `while` utasítás általánosítása. Ha összehasonlítjuk a korábban használt `while` utasítással, a `for` működése teljesen világos. A zárójelek között három, egymástól pontosvesszővel elválasztott rész található. Az első, kezdeti értékadó rész

```
fahr = 0
```

amit csak egyszer hajt végre a program, a ciklusba való belépés előtt. A második rész a ciklust vezérlő ellenőrzés vagy feltétel

```
fahr <= 300
```

alakú. Működés közben a gép megvizsgálja ezt a feltételt, és ha igaz, akkor végrehajtja a ciklusmagot (ami most csak egyetlen `printf` utasítás). Ezután történik a lépésközzel való növelés

```
fahr = fahr + 20
```

amit a feltétel újbóli ellenőrzése követ. A ciklus akkor fejeződik be, ha a feltétel értéke hamis lesz. Csakhogy, mint a `while` utasításnál, a ciklusmag itt is lehet egyetlen utasítás vagy kapcsos zárójelek között elhelyezett utasításcsoport. A kezdeti értékadás, feltételvizsgálat és a lépésközzel való növelés tetszőleges kifejezéssel adható meg.

A `while` és a `for` között szabadon választhatunk aszerint, hogy számunkra melyik tűnik világosabbnak. A `for` utasítás általában akkor előnyös, ha a kezdeti értékadás és lépésközzel való növelés egy-egy logikailag összefüggő utasítás, mivel ekkor a `for` utasítással szervezett ciklus sokkal tömörebb a `while` utasítással szervezett ciklusnál és a ciklust vezérlő utasítások egy helyen vannak.

1.5. gyakorlat. Módosítsuk a hőmérséklet-átalakító programot úgy, hogy a táblázatot fordított sorrendben, tehát 300 foktól 0 fokig nyomtassa ki!

1.4. Szimbolikus állandók

Mielőtt elbúcsúznánk a hőmérséklet-átalakító programunktól, még egy észrevételt teszünk: nagyon rossz gyakorlat a 300-hoz vagy a 20-hoz hasonló „bűvös számokat" beépíteni a programba. Ezek később, a program olvasásakor nem sokat mondanak és az esetleges megváltoztatásuk nagyon nehézkes. Az ilyen bűvös számok kiküszöbölésének egyik módja, hogy „beszélő" neveket rendelünk hozzájuk. A `#define` szerkezet lehetővé teszi, hogy egy megadott karakterlánchoz szimbolikus nevet vagy szimbolikus állandót rendeljünk, az alábbiak szerint:

`#define` *név helyettesítő szöveg*

Ezután a fordítóprogram a *név* minden önálló előfordulásakor (amikor a név nincs idézőjelek közt vagy nem része egy másik névnek) a név helyett a megadott *helyettesítő szöveget* írja be. A névre ugyanazok a szabályok vonatkoznak, mint a változók nevére: betűkből, ill. számjegyekből állhat, és betűvel kell kezdődnie. A helyettesítő szöveg tetszőleges karaktersorozat lehet, nem csak szám. A `#define` használatát a következő példán mutatjuk be.

```
#include <stdio.h>

#define ALSO 0     /* a táblázat alsó határa */
#define FELSO 300  /* a táblázat felső határa */
#define LEPES 20   /* a táblázat lépésköze */

/* a Fahrenheit-fok-Celsius-fok táblázat kiírása */
main( )
{
    int fahr;

    for (fahr = ALSO; fahr <= FELSO; fahr = fahr + LEPES)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Az ALSO, FELSO, LEPES szimbolikus állandók és nem változók, így nem szerepelnek a deklarációkban. A szimbolikus neveket általában nagybetűkkel írjuk (de ez nem kötelező), hogy megkülönböztethetők legyenek a kisbetűvel írt változónevektől. Megjegyezzük, hogy a define szerkezet végén nincs pontosvessző.

1.5. Karakteres adatok bevitele és kivitele

A következőkben néhány egymással összefüggő, karakteres adatok feldolgozására alkalmas programot ismertetünk. A későbbiekben látni fogjuk, hogy számos bonyolult program ezeknek a példaprogramoknak a kibővített változata.

A karakteres adatok be- és kivitelének standard könyvtárral támogatott megvalósítása nagyon egyszerű. Szövegek be- és kivitelét – függetlenül attól, hogy honnan erednek vagy hová irányulnak – karakterek áramaként fogjuk fel. A *szövegáram* legalább két sorból álló karakteráram (karaktersorozat), amelynek mindegyik sora nulla vagy annál több karakterből áll és a végén egy újsor-karakter helyezkedik el. A standard könyvtár feladata, hogy az adatáramok be- és kivitelét a fenti modell alapján kezelje. A C nyelvet használó programozó ezeket a könyvtári függvényeket használja, és nem törődik azzal, hogy az egyes sorok a programon kívül mit jelentenek.

A standard könyvtárban számos olyan függvény van, amelyekkel egy időben egy karakter olvasható vagy írható, és ezen függvények közül a legegyszerűbb a `getchar` és `putchar` függvény. Minden egyes hívásakor a `getchar` függvény a szövegáramból beolvassa a következő karaktert és annak értékét adja vissza a hívó függvénynek. Ennek megfelelően a

```
c = getchar( )
```

végrehajtása után a `c` változó a bemenő szöveg következő karakterét fogja tartalmazni. A karakterek általában a terminálról (billentyűzetről) érkeznek, az adatállományból történő beolvasással a 7. fejezetben fogunk foglalkozni.

A `putchar` függvény minden egyes hívásakor kiír egy karaktert. A

```
putchar(c)
```

végrehajtása során a `c` egész típusú változó tartalma mint egy karakter íródik ki,

általában a képernyőre. A `putchar` és a `printf` hívások felváltva is történhetnek, ilyenkor a kimenet a hívások sorrendjében fog megjelenni.

1.5.1. Állománymásolás

A `getchar` és `putchar` felhasználásával nagyon sok programot írhatunk a bemenet és a kimenet pontos ismerete nélkül. A legegyszerűbb ilyen mintaprogram a bemenetet karakterenként átmásolja a kimenetre. A program szerkezete:

egy karakter beolvasása
`while` (*a karakter nem az állományvége-jel*)
    az éppen beolvasott karakter kimenetre írása
    egy új karakter beolvasása

Mindez C nyelvű programként megfogalmazva:

```
#include <stdio.h>

/* a bemenet átmásolása a kimenetre - 1. változat */
main( )
{
    int c;

    c = getchar( );
    while (c != EOF) {
        putchar(c);
        c = getchar( );
    }
}
```

A `while` utasításban szereplő `!=` operátor jelentése „nem egyenlő”.

A billentyűzeten vagy képernyőn megjelenő karakter a számítógépen belül bitmintaként tárolódik. A `char` típus egy ilyen karakteres adat tárolására alkalmas tárolóhelyet specifikál, de erre a célra bármilyen egész típusú adat is alkalmas. A programban a karakter tárolására `int` típusú változót használtunk egy bonyolult, de lényeges okból.

A program működése során a fő probléma az érvényes bemeneti adatok végének érzékelése. A probléma úgy oldható meg, ha a `getchar` függvény egy olyan értékkel tér vissza a bemeneti adatok elfogyása esetén, ami semmilyen más, létező karakterrel nem téveszthető össze. Ezt az értéket `EOF`-nak, (*end of file*), állományvége-jelnek nevezik. A programban a `c` változót úgy kellett deklarálni, hogy elegendően nagy legyen bármilyen, a `getchar` által visszaadott érték tárolására. Ezért a `c` változó nem lehet `char` típusú, mivel az nem elegendően nagy az `EOF` befogadására, így `int` típusú változót használunk.

Az `EOF` az `<stdio.h>` headerben definiált egész érték, amelynek konkrét értéke mindaddig nem lényeges, amíg különbözik bármely `char` típusú értéktől. Az `EOF` szimbolikus állandó használatával garantálható, hogy egyetlen program működése sem függ az `EOF` tényleges számértékétől.

A gyakorlottabb C programozók a fenti másolóprogramot sokkal tömörebben írnák meg. A C nyelvben a

```
c = getchar( )
```

jellegű értékadások kifejezésekbe is beépíthetők és értékük megegyezik az értékadás bal oldalának értékével. Ez azt jelenti, hogy egy értékadás egy nagyobb kifejezés része lehet. Ha az egy karaktert a c változóhoz rendelő értékadást a while ciklus ellenőrző részébe építjük be, akkor a másoló program a következő módon fog kinézni:

```
#include <stdio.h>

/* a bemenet átmásolása a kimenetre - 2. változat */
main( )
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

A program a while ciklusban beolvas egy karaktert, hozzárendeli a c változóhoz, majd megvizsgálja, hogy a beolvasott karakter megegyezik-e az állományvége-jellel. Ha nem, akkor a while ciklusmagja végrehajtódik és a karakter kiíródik, majd a ciklusmag ismétlődik. Amikor a bemeneti karaktersorozat véget ér, a while ciklus befejeződik és ezzel együtt a main is.

A programnak ez a változata egy helyre koncentrálja az adatbeolvasást – csak egyszer szerepel benne a getchar függvény hívása –, így rövidebbé, ill. olvashatóbbá válik a program. Ezzel a programozási stílussal gyakran találkozunk. Bár a túlzott tömörítésnek megvan az a veszélye, hogy a program áttekinthetetlenné válik, ezt a továbbiakban is igyekszünk elkerülni.

A while utasításban az értékadás körüli zárójelek feltétlenül szükségesek, mivel a != operátor *precedenciája* nagyobb, mint az = operátoré és ezért a zárójelek hiányában először a reláció kiértékelése történne meg, és csak ezt követné az értékadás. A zárójelet elhagyva a

```
c = getchar( ) != EOF
```

utasítást kapjuk, ami egyenértékű a

```
c = (getchar( ) != EOF)
```

utasítással, aminek nem kívánt következménye, hogy a c változóhoz a 0 vagy 1 értéket rendeli attól függően, hogy a getchar az állományvége-jelet olvasta-e vagy sem. A kérdéskörrel a 2. fejezetben még részletesen foglalkozunk.

1.6. gyakorlat. Igazoljuk, hogy a getchar( ) != EOF kifejezés értéke valóban 0 vagy 1!

1.7. gyakorlat. Írjunk programot, ami kiírja az EOF értékét!

1.5.2. Karakterek számlálása

A következő példaprogram a másolóprogramhoz hasonlóan működik és megszámlálja a beolvasott karaktereket.

```
#include <stdio.h>

/* a beolvasott karaktereket számláló program */
/* 1. változat */
main( )
{
    long nc;

    nc = 0;
    while (getchar( ) != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

A programban szereplő

```
++nc;
```

utasításban egy új operátor, a ++ található, amelynek jelentése: növelj eggyel *(inkrementálás)*. Ehelyett természetesen azt is írhatnánk, hogy `nc = nc+1`, de a `++nc` sokkal tömörebb és gyakran hatékonyabb is. Létezik a -- operátor is, ami az eggyel való csökkentést *(dekrementálás)* valósítja meg. A ++ és -- operátor egyaránt lehet *előtag (prefix)* és *utótag (postfix)* operátor (`++nc`, ill. `nc++` vagy `--nc`, ill. `nc--`). A kétféle forma a kifejezésben különböző értéket ad, ennek pontos leírásával a 2. fejezetben találkozunk, de a lényeg az, hogy mind a `++nc`, mind az `nc++` növeli az `nc` értékét. Egyelőre mi a prefix formát használjuk.

A karaktereket számláló program a kapott számot `int` helyett `long` típusú változóban tárolja. Az `int` típusú változók max. értéke 32 767 lehet, ami viszonylag kicsi, és számlálóként `int` típusú változót használva hamar túlcsordulás jelentkezne. A `long` típusú egész számot a legtöbb számítógép legalább 32 biten ábrázolja (bár néhány számítógépen az `int` és a `long` típusú változók egyaránt 16 bitesek). A `%ld` konverziós specifikáció azt jelzi a `printf` függvénynek, hogy a megfelelő argumentum `long` típusú egész szám.

Sokkal nagyobb számokig is elszámlálhatnánk, ha `double` (kétszeres pontosságú lebegőpontos) változót használnánk.

A ciklusszervezés másik módjának szemléltetésére a `while` helyett használjuk a `for` utasítást.

```
#include <stdio.h>

/* a beolvasott karaktereket számláló program */
/* 2. változat */
main( )
{
    double nc;

    for (nc = 0; getchar( ) != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

A kiíratásban a `float` és `double` típusú adatokhoz egyaránt használhatjuk a `%f` specifikációt, és a `%.0f` specifikáció elnyomja a tizedespont és a törtrész kiírását (a törtrész hossza nulla jegy).

A ciklusmag üres, hiszen minden műveletet az ellenőrző és növelő részben végzünk el. A C nyelv szintaktikai (nyelvtani) szabályai viszont megkövetelik, hogy a `for` utasítással szervezett ciklusnak legyen magja. Az önmagában álló pontosvessző, azaz *nulla* (vagy *üres*) *utasítás* ezt a követelményt kielégíti. Az üres utasítást külön sorba írtuk, hogy kihangsúlyozzuk a fontosságát.

Mielőtt befejeznénk a karaktereket számláló program tárgyalását, felhívjuk a figyelmet arra, hogy ha a bemeneten egyáltalán nincs adat, akkor a `getchar` első hívása után a `while` vagy a `for` vizsgáló része hamis eredményt ad, a program nulla számú karaktert számol, azaz helyesen működik, ami nagyon fontos. A dolog a `while` és a `for` azon kedvező tulajdonságával kapcsolatos, hogy mindkettő a ciklus elején, a ciklusmag végrehajtása előtt ellenőrzi a feltételt (elöltesztelő ciklus). Ha tehát semmit sem kell csinálni, akkor a ciklus valóban nem csinál semmit, még akkor sem, ha emiatt soha nem hajtja végre a ciklusmagot. A programjainknak határesetben (ha a bemeneten nulla hosszúságú adatsor van) is helyesen kell működniük.

1.5.3. Sorok számlálása

A következő példaprogramunk megszámolja a bemenetre adott adatsorokat. Mint korábban már említettük, a standard könyvtár a bemeneti szövegáramot egymást követő sorok sorozataként értelmezi és minden sort az újsor-jel zár. Ebből következik, hogy a sorok számlálása lényegében az újsor-karakterek számlálásának felel meg. Így a program:

```
#include <stdio.h>

/* a bemenő szöveg sorainak számlálása */
main( )
{
    int c, nl;

    nl = 0;
    while ((c = getchar( )) != EOF)
        if (c == '\n')

            ++nl;
    printf("%d\n", nl);
}
```

A `while` ciklusmagja most egy `if` utasítást tartalmaz, amely a `++nl` inkrementáló utasítás végrehajtását vezérli. Az `if` utasítás ellenőrzi a zárójelben lévő feltételt, és ha az igaz, akkor végrehajtja a következő utasítást (vagy a kapcsos zárójelek közt elhelyezett utasításcsoportot). A program elrendezése most is világossá teszi, hogy mi mit vezérel.

Az == kettős egyenlőségjel jelentése a C nyelvben az „egyenlő valamivel" (hasonlóan a Pascal = jeléhez vagy a FORTRAN `.EQ.` operátorához). Az == szimbólumot azért vezették be, hogy az egyenlőség vizsgálatát megkülönböztessék az = jellel jelölt értékadástól. Még egy figyelmeztető megjegyzés: a kezdő C programozók gyakran írnak = jelet ott, ahol == kellene. Amint ezt a 2. fejezetben látni fogjuk, az eredmény általában egy érvényes (de az adott helyen értelmetlen) kifejezés, így a fordítóprogram nem ad hibajelzést.

Két aposztróf között elhelyezett karakter egy egész számot jelent, amelynek értéke a karakter gépi karakterkészletbeli kódjával egyezik meg. Az aposztrófok között lévő karaktert *karakterállandónak* nevezzük és helyette használhatjuk a neki megfelelő kis egész számot (kódot) is. Például az `'A'` egy karakterállandó, amelynek értéke az ASCII karakterkészletben 65, vagyis ez a szám az `A` belső, gépi ábrázolása. Természetesen `'A'` helyett 65-öt is írhatnánk, de az `'A'` jelentése sokkal világosabb és független az éppen használt karakterkészlettől.

Az escape sorozatok karakterállandókénti megadása szintén lehetséges, így a `'\n'` az újsor-karakter értékét jelenti, ami az ASCII karakterkészletben 10. Ne feledjük, hogy a `'\n'` egyetlen karakter és a kifejezésekben egyetlen számnak felel meg, az `"\n"` pedig egy karaktersorozat (string), ami adott esetben csak egy karaktert tartalmaz. A karakterek és karaktersorozatok témájával szintén a 2. fejezetben foglalkozunk majd részletesebben.

1.8. gyakorlat. Írjunk programot, ami megszámolja a bemenetre adott szövegben lévő szóközöket, tabulátorokat és újsor-karaktereket!

1.9. gyakorlat. Írjunk programot, ami a bemenetre adott szöveget úgy másolja át a kimenetre, hogy közben az egy vagy több szóközből álló karaktersorozatokat egyetlen szóközzel helyettesíti!

1.10. gyakorlat. Írjunk programot, ami a bemenetre adott szöveget úgy másolja át a kimenetre, hogy közben a tabulátorkaraktereket `\t`, a visszaléptetés- (backspace) karaktereket `\b` és a fordított törtvonal- (backslash) karaktereket `\\` karakterekkel helyettesíti! Ezzel az átírással a tabulátor- és visszaléptetés-karakterek a nyomtatásban is láthatóvá válnak.

1.5.4. Szavak számlálása

A sorozat negyedik programja a sorokon és a karaktereken kívül megszámolja a bemenetre adott szövegben lévő szavakat is. A szó számunkra olyan tetszőleges karaktersorozatot jelent, amelyben nem fordul elő a szóköz-, tabulátor- vagy újsor-karakter. A szó fenti, elég laza definíciója alapján működő program (ami a UNIX `wc` segédprogramjának váza) a következő:

```
#include <stdio.h>

#define BENN 1 /* a szó belseje */
#define KINT 0 /* a szón kívül */

/* a bemenetre adott szövegben lévő sorok,
szavak és karakterek számolása */
main( )
{

    int c, nl, nw, nc, allapot;

    allapot = KINT;
    nl = nw = nc = 0;
    while ((c = getchar( )) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
```

```
        if (c == ' ' || c == '\n' || c == '\t')
            allapot = KINT;
        else if (allapot == KINT) {
            allapot = BENN;
            ++nw;
        }
    }
    print("%d %d %d\n", nl, nw, nc);
}
```

Amint a program megtalálja egy szó első karakterét, azonnal növeli a szószámlálót (nw). Az `allapot` változó azt jelzi, hogy pillanatnyilag a szó belsejében vagyunk-e vagy sem. Kezdetben „nincs a szóban", amit a hozzárendelt `KINT` érték jelez. A programban a `BENN` és a `KINT` *szimbolikus állandók* használata előnyösebb az 1 és 0 értékeknél, mivel a program olvashatóbbá válik. Az olvashatóság ebben a kis példában nem okoz nehézséget, de nagy programoknál az áttekinthetőség növekedése lényegesen több hasznot jelent. Az olyan program, amiben a „bűvös számok" helyett szimbolikus állandókat használunk, könnyebben is módosítható.

A program

```
nl = nw = nc = 0;
```

sorában mindhárom változóhoz nulla értéket rendelünk. Ez nem egy speciális utasításfajta, hanem abból következik, hogy az értékadásban egy kifejezés szerepel valamilyen értékkel és az értékadások jobbról balra haladva hajtódnak végre. Az előzővel teljesen egyenértékű az

```
nl = (nw = (nc = 0));
```

értékadás.

A `||` operátor a logikai VAGY műveletet jelenti, így a

```
if (c == ' ' || c == '\n' || == '\t')
```

utasítás azt jelenti, hogy „ha `c` szóköz *vagy* `c` újsor *vagy* `c` tabulátor, akkor...". (Mint korábban már említettük, a `\t` escape sorozat a tabulátorkaraktert jelzi.) Van egy másik logikai operátor is, az `&&`, ami a logikai ÉS műveletet jelenti és ennek precedenciája nagyobb, mint a logikai VAGY műveleté. Az `&&` és `||` operátorokkal összekapcsolt kifejezések kiértékelése balról jobbra történik, és a kiértékelés azonnal félbeszakad, ha a kifejezés igaz vagy hamis volta egyértelművé válik. Ha a `c` szóköz volt, akkor nincs értelme azt vizsgálni, hogy `c` tartalma újsor- vagy tabulátorkarakter-e, így ezek a vizsgálatok már nem mennek végbe. Ez itt most nem különösen fontos, de bonyolultabb esetekben, amint azt hamarosan látni fogjuk, nagyon lényeges lehet.

A példában előfordul az `else` utasítás, ami meghatározza a program működését abban az esetben, ha az `if` utasítás feltétele hamis volt. Az utasítás általános formája:

```
if (kifejezés)
    1. utasítás
else
    2. utasítás
```

Az `if-else` szerkezetnek mindig csak az egyik utasítása hatásos. Ha a *kifejezés* igaz, akkor az *1. utasítást,* ha nem, akkor pedig a *2. utasítást* hajtja végre a program. Az *1.* és *2. utasítások* önálló utasítások vagy kapcsos zárójelben elhelyezett utasításcsoportok lehetnek. A szavakat számláló programban az `else` után egy `if` utasítás áll, ami két másik, kapcsos zárójelbe foglalt utasítást vezérel.

1.11. gyakorlat. Hogyan lehet ellenőrizni a szavakat számláló programot? Milyen bemeneti adatsort kell használni, hogy a legnagyobb valószínűséggel érzékeljük a program esetleges hibáit?

1.12. gyakorlat. Írjunk programot, ami a bemenetére adott szöveg minden szavát új sorba írja ki!

1.6. Tömbök

Írjunk programot, amely megszámlálja, hogy a bemenetre adott szövegben hányszor fordulnak elő az egyes számjegyek, az üres helyet jelentő karakterek (szóköz, tabulátor, új sor), valamint az összes többi karakter! Ez egy elég mesterkélt feladat; de lehetővé teszi, hogy egyetlen programban jól szemléltessük a C nyelv számos lehetőségét.

Mivel a bemeneti adatokat 12 kategóriába kell sorolni, kézenfekvőnek látszik az egyes számjegyek előfordulásainak számát egy tömbben tárolni, tíz külön változó helyett. Ennek megfelelően a program egyik változata a következő:

```
#include <stdio.h>

/* számjegyeket, üres helyeket és más
karaktereket számláló program*/
main( )
{
    int c, i, nures, nmas;
    int ndigit[10];

    nures = nmas = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nures;
        else
            ++nmas;

    printf("számok =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", üres = %d, más = %d\n",
        nures, nmas);
}
```

A program az eredményt pl. a következő módon írja ki:

```
számok = 9 3 0 0 0 0 0 0 0 1, üres = 123, más = 345
```

Az

```
int ndigit[10];
```

deklaráció a 10 egész számot tartalmazó `ndigit` tömböt deklarálja. A C nyelvben a tömbök indexe mindig nullától indul, így az `ndigit`-nek `ndigit[0]`, `ndigit[1]`, `...`, `ndigit[9]` elemei vannak. Ez tükröződik a tömböt kezdő értékkel feltöltő (inicializáló) és kinyomtató `for` ciklusokban.

Az index tetszőleges egész típusú kifejezés lehet, ami megadható egy egész típusú változóval (pl. `i`) vagy egész típusú állandóval.

A példaprogram nagymértékben kihasználja a számjegyek karakteres ábrázolásának tulajdonságait. Így pl. az

```
if (c >= '0' && c <= '9') ...
```

vizsgálat eldönti, hogy a `c` változóban lévő karakter számjegy-e. Ha az, akkor ennek a számjegynek a numerikus értéke

```
c - '0'
```

Ez a módszer természetesen csak akkor alkalmazható, ha a `'0'`, `'1'`, `...`, `'9'` egymást követő, növekvő sorrendű, pozitív egész számok. Szerencsére ez minden karakterkészlet esetén teljesül.

Definíció szerint a `char` típusú adatok kis egész számok, ezért az aritmetikai kifejezésekben a `char` és `int` típusú változók és állandók egyenértékűek. Ez elég természetes és kényelmes megoldás, pl. a `c-'0'` egész kifejezés értéke 0 és 9 közt van, attól függően, hogy a `'0'...'9'` karakterek közül melyik tárolódott a `c` változóban. Az így kapott érték felhasználható az `ndigit` tömb indexeként.

Annak eldöntése, hogy a karakter számjegy, üres hely vagy valami más-e, a következő vizsgálatsorozattal lehetséges:

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nures;
else
    ++nmas;
```

Az

```
if (1. feltétel)
    1. utasítás
else if (2. feltétel)
    2. utasítás
...
    ...
else
    n. utasítás
```

felépítésű szerkezetek gyakran előfordulnak a programokban, mint a többutas elágazások megvalósításai. A gép a *feltételeket* felülről kezdi kiértékelni és ha az igaz, akkor végrehajtja a megfelelő *utasítást,* majd befejezi a szerkezetet. (Természetesen bármelyik utasítás helyett kapcsos zárójelben elhelyezett utasításcsoport is állhat.) Ha egyik feltétel sem teljesül, akkor az utolsó `else` utáni *utasítást* hajtja végre. Ha az utolsó `else` és a hozzá tartozó *utasítás* hiányzik (mint a szavakat számláló programban), akkor semmi nem történik. Egy programban a kezdeti if és a végső else között tetszőleges számú

```
else if (feltétel)
    utasítás
```

felépítésű utasításcsoport lehet.

Stilisztikai szempontból célszerű a példaprogramban bemutatott formát követni, mert így a programsorok nem lesznek túl hosszúak, a hosszú döntési láncok nem nyúlnak a jobb margón túlra.

A 3. fejezetben fogjuk ismertetni a `switch` utasítást, ami szintén a többutas elágazások leírására alkalmas. Ez főleg akkor használható előnyösen, ha azt vizsgáljuk, hogy egy egész vagy karakteres típusú kifejezés egyezik-e egy állandókból álló halmaz valamelyik elemével. Összehasonlítás céljából a 3.4. pontban bemutatjuk a példaprogramunk `switch` utasítással megvalósított változatát.

1.13. gyakorlat. Írjunk programot, ami kinyomtatja a bemenetre adott szavak hosszának hisztogramját! A legcélszerűbb, ha a hisztogramot vízszintesen ábrázoljuk, mert a függőleges ábrázolás túl bonyolult lenne.

1.14. gyakorlat. Írjunk programot, ami kinyomtatja a bemenetre adott különböző karakterek előfordulási gyakoriságának hisztogramját!

1.7. Függvények

A C nyelv függvényei megfelelnek a FORTRAN szubrutinjainak vagy függvényeinek, vagy a Pascal eljárásainak vagy függvényeinek. A függvény kényelmes lehetőséget nyújt a programozónak, hogy egy számítási részt önállóan kezelhető, zárt egységbe foglaljon. Ezek a zárt egységek ezután szabadon felhasználhatók anélkül, hogy a konkrét felépítésükkel, megvalósításukkal törődnünk kellene. Megfelelően tervezett és megvalósított függvények esetén teljesen figyelmen kívül hagyhatjuk, hogy *hogyan* keletkezett a függvény értéke (eredménye), elegendő csak az *eredményt* tudni. A C nyelvben a függvények használata egyszerű, kényelmes és hatékony. Gyakran találkozunk majd rövid, néhány sorban definiált és csak egyszer meghívott függvényekkel, amelyeknek pusztán csak az a szerepe, hogy a programot áttekinthetővé tegyék.

Ez idáig csak kész, könyvtári függvényekkel (`printf, getchar, putchar`) találkoztunk, így itt az ideje, hogy magunk is írjunk függvényeket. Mivel a C nyelvnek nincs a FORTRAN-ban értelmezett `**`-hoz hasonló hatványozó operátora, ezért a függvénydefiniálás bemutatására írjuk meg a `power(m, n)` függvényt, amely előállítja egy `m` egész szám `n`-edik hatványát (`n` pozitív egész szám). Például a `power(2, 5)` értéke 32 lesz. A példaként választott függvény nem egy valódi hatványozó eljárás, mivel csak kis egész számok pozitív, egész kitevős hatványait képes kiszámítani. (A standard könyvtár `pow(x, y)` függvénye egy általános, x^y alakú kifejezés értékét határozza meg.)

A következőkben bemutatjuk a `power` függvényt és az azt hívó `main` főprogramot (ami maga is függvény), ami alapján a teljes szerkezet elemezhető.

```
#include <stdio.h>

int power(int m, int n);

/* a hatványozó függvény ellenőrzése */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2, i), power(-3, i));
    return 0;
}
/* a power(m, n) függvény az m alapot az n-edik hatványra
emeli, ahol n >= 0 */
int power(int alap, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * alap;
    return p;
}
```

A függvénydefiníció általános alakja:

visszatérési típus *függvénynév (paraméter-deklarációk, ha vannak)*
{
deklarációk
utasítások
}

A függvénydefiníciók tetszőleges sorrendben szerepelhetnek, egy vagy több forrásállományban (bár természetesen egy függvény nem osztható szét két forrásállományba). Ha a forrásprogram több állományban helyezkedik el, bonyolultabb a fordítás és a betöltés, mintha egyetlen állományban lenne, de ez az operációs rendszer problémája és nem a nyelv jellegzetessége. Pillanatnyilag feltételezzük, hogy mindkét függvény azonos állományban van, így a C programok futtatásáról elmondottak továbbra is érvényesek.

A `main` a `power` függvényt kétszer hívja a

```
printf("%d %d %d\n", i, power(2, i), power(-3, i));
```

sorban. Mindegyik híváskor két argumentumot adunk át a `power` függvénynek és az egy egész értékkel tér vissza, amit a hívó program formátumozott formában kiír. Egy aritmetikai kifejezésben a `power(2, i)` éppen olyan egész mennyiség, mint `2` és `i`. (Nem minden függvény ad egész értéket, erről majd a 4. fejezetben mondunk többet.)

A `power` függvény első sorában az

```
int power(int alap, int n)
```

a paraméterek (argumentumok) típusát és nevét, valamint a függvény visszatéréskor szolgáltatott értékének típusát deklarálja. A `power` függvény által használt paraméterek nevei a `power` függvényre nézve lokálisak, azaz egyetlen más függvényben sem „láthatók", bármely eljárás ugyanezeket a neveket minden gond nélkül saját célra használhatja. Ez szintén igaz a `power`-ben deklarált `i` és `p` változókra is, és a `power`-ben használt `i` változónak semmi köze a `main`-ben használt `i` változóhoz.

A fogalmak egyértelművé tétele érdekében a továbbiakban *paraméternek* nevezzük a függvénydefiníció során zárójelben megadott változóneveket, és *argumentumnak* a függvény hívása során átadott értékeket. Ezekre a fogalmakra néha a *formális* és *aktuális argumentum* vagy *formális* és *aktuális paraméter* fogalmakat is használják.

A `power` függvényben kiszámított értéket a `return` utasítás adja vissza a `main` függvénynek. A `return` utasítás után tetszőleges kifejezés állhat a

```
return kifejezés;
```

szintaktika szerint. Egy függvénynek nem feltétlenül szükséges értékkel visszatérni a hívó programba. Egy kifejezés nélküli `return` utasítás visszaadja a vezérlést a hívó programnak, de nem ad át hasznos információt. Ez történik akkor is, ha a vezérlés átlépi a függvény végét jelző jobb oldali kapcsos zárójelet. Természetesen a hívó függvénynek jogában áll nem figyelembe venni a hívott függvény által visszaadott értéket.

Vegyük észre, hogy a `main` végén is van egy `return` utasítás. Mivel a `main` ugyanolyan függvény, mint bármelyik más, ezért visszaadhat egy értéket a hívó programnak. A `main` hívó programja az a környezet, amiben a program lefut, és a visszaadott értéket ez a környezet használja fel. Tipikusan a nulla visszaadott érték a program normális lefutását jelzi, a nullától különböző érték pedig a program abnormális vagy hibás lefutására utal. Az egyszerűség kedvéért eddig elhagytuk a `return` utasítást a `main` függvényeinkből, de ezután használni fogjuk, hogy a programunk egy állapotjelzést adhasson a környezetének.

Az

```
int power(int m, int n);
```

deklaráció a `main` elején azt mondja meg, hogy a `power` függvény két egész típusú argumentumot vár és egész típusú eredménnyel tér vissza. Ezt a deklarációt *függvényprototípusnak* nevezik, és meg kell hogy egyezzen a `power` függvény definíciójával és használatával. Programozási hiba, ha a függvény definíciója vagy bármilyen használata nem egyezik a függvényprototípussal.

Természetesen a paraméterek neveinek nem kell egyezni, a függvény prototípusaiban a paraméternevek opcionálisak és a `power` prototípusát így is írhatnánk:

```
int power(int, int);
```

Mindenesetre a jól választott nevek jó programdokumentálást tesznek lehetővé, ezért a továbbiakban is gyakran használni fogjuk a neveket a prototípusban.

Végezetül egy történelmi megjegyzést szeretnénk tenni: az ANSI C és a korábbi C változatok közti legnagyobb eltérés éppen a függvények definiálásában és deklarálásában tapasztalható. A C nyelv eredeti definícója szerint a *power* függvényt a következő módon írtuk volna meg:

```
/* a power(m, n) függvény az m alapot az n-edik hatványra
emeli, ahol n >= 0 - régi típusú változat */
power(alap, n)
int alap, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * alap;
    return p;
}
```

Itt a paraméterek nevét zárójelek között adtuk meg és a típusaikat a nyitó kapcsos zárójel előtt deklaráltuk. A deklarálatlan paraméterek `int` típusúak lesznek. (A függvény törzse mindkét változatban megegyezik.)

A `main` kezdetén a `power` függvényt az

```
int power( );
```

utasítással deklaráltuk volna. Itt nincs megengedve a paraméterlista, így a fordítóprogram nem képes egyszerű módon ellenőrizni, hogy a `power` függvényt helyesen hívták-e. Valójában kiindulhatnánk az alapfeltételezésből is, és mivel a `power int` típusú értékkel tér vissza, így az egész deklarációt elhagyhatnánk.

A függvényprototípusok új szintaktikája sokkal egyszerűbbé teszi a fordítóprogram számára az argumentumok számában és típusában elkövetett hibák észlelését. A függvények régi típusú definiálási és deklarálási módja átmenetileg még érvényben van az ANSI C-ben, de javasoljuk, hogy mindenki az új változatot használja, ha a fordítóprogramja támogatja azt.

1.15. gyakorlat. Írjuk át az 1.2. pontban ismertetett hőmérséklet-átalakító programot úgy, hogy az átalakításhoz függvényt használunk!

1.8. Argumentumok – az érték szerinti hívás

A C nyelv függvényeinek egyik tulajdonságát a más nyelvekben – különösen FORTRAN nyelvben – gyakorlatot szerzett programozók szokatlannak fogják találni: a C nyelvben minden függvényargumentumot érték szerint adunk át. Ez azt jelenti, hogy a hívott függvény mindig az argumentumok értékét kapja meg (átmeneti változókban), azok címe helyett. Ez sok esetben eltérést okoz az olyan nyelvekhez képest, ahol a hívott eljárás az eredeti argumentum címét kapja meg egy helyi másolat helyett (mint pl. a FORTRAN-ban, ahol név szerinti a paraméterátadás, vagy mint pl. a Pascalban, ahol a var deklaráción keresztüli paraméterátadás van).

A legfontosabb különbség ezekhez a nyelvekhez képest, hogy a C nyelvben a hívott függvény közvetlenül nem férhet hozzá és nem változtathatja meg a hívó függvény változóit, csak a saját, átmeneti másolatával dolgozhat.

Az érték szerinti hívás mindenképpen előny. Felhasználásával sokkal tömörebb, kevesebb segédváltozót tartalmazó program írható, mivel a hívott függvényben a paraméterek ugyanúgy kezelhetők, mint az inicializált lokális változók. Az elmondottak illusztrálására nézzük a `power` függvény egy újabb változatát!

```
/*a power(m, n) függvény az m alapot az n-edik hatványra
emeli, ahol n >= 0 - 2. változat */
int power(int alap, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * alap;
    return p;
}
```

A programban az n paramétert átmeneti változóként használtuk és lefelé számláltattuk (a for ciklus csökkenő irányú), amíg csak nulla nem lett. Ennek következtében nincs szükség az i segédváltozóra. Az n értékének power függvényen belüli módosítása semmilyen hatással sincs a híváskor használt eredeti értékre.

Szükség esetén természetesen elérhető, hogy a hívott függvény megváltoztassa a hívó függvény valamelyik változóját. Ehhez a hívott függvénynek meg kell kapnia a kérdéses változó *címét* a hívó függvénytől (ami a változót megcímző *mutató* átadásával lehetséges) és a hívott függvényben a paramétert mutatóként kell deklarálni, amelyen keresztül indirekten érhető el a hívó függvény változója. A kérdéssel az 5. fejezetben fogunk részletesebben foglalkozni.

A leírtak nem érvényesek a tömbökre. Ha argumentumként egy tömb nevét adjuk meg, akkor a függvénynek átadott érték a tömb kezdetének helye (címe) lesz, a tömbelemek átmásolása nem történik meg. Ezt az értéket indexelve a hívott függvény a hívó függvény bármelyik tömbeleméhez hozzáférhet. Ezzel a kérdéssel a következő pontban foglalkozunk.

1.9. Karaktertömbök

A C nyelvben valószínűleg a leggyakrabban használt tömbtípus a karaktertömb. Annak bemutatására, hogy hogyan használjuk a karaktertömböket, ill. hogyan manipuláljuk azokat a megfelelő függvényekkel, írjunk egy programot, ami szövegsorokat olvas be és kiírja közülük a leghosszabbat. A program váza viszonylag egyszerű:

```
while  (van további sor)
    if  (a sor hosszabb az eddigi leghosszabbnál)
        tárold a sort
        tárold a sor hosszát
nyomtasd ki a leghosszabb sort
```

A programváz alapján látszik, hogy a program több, jól elkülönülő részre osztható: az első beolvassa, és megvizsgálja, a második pedig eltárolja az új sort, a fennmaradó rész pedig a folyamatot vezérli.

Mivel az egyes részek ilyen jól elkülönülnek, célszerű, ha a programot is ennek megfelelően írjuk meg. Ezért először írjunk egy önálló getline függvényt, aminek feladata, hogy előkészítse a következő sort. Megpróbáljuk a getline függvényt úgy megírni, hogy az más programokban is jól használható, kellően általános legyen. A minimális igény, hogy a getline visszatérésekor jelezze, ha elértük az állomány végét. Sokkal általánosabbá tehetjük a függvényt, ha az visszatéréskor a sor hosszát adja meg, vagy nullát, ha elértük az állomány végét. A nulla hossz biztosan megfelel

az állomány végének jelzésére, mivel nem lehet tényleges sorhossz (minden szövegsor legalább egy karaktert, az újsor-karaktert tartalmazza, így a hossza minimálisan 1).

Ha találunk egy sort, ami hosszabb az eddigi leghosszabb sornál, akkor azt valahová el kell tennünk. Erre a célra egy másik, `copy` nevű függvényt használunk, amelynek feladata az új sor biztos helyre mentése.

Végül szükségünk van egy `main`-re a `getline` és a `copy` vezérléséhez. Az eredmény a következő:

```
#include <stdio.h>
#define MAXSOR 1000 /* a beolvasott sor max. mérete */

int getline(char sor[ ], int maxsor);
void copy(char ba[ ], char bol[ ]);

/* a leghosszabb sor kiíratása */
main( )
{
    int hossz;              /* az aktuális sor hossza */
    int max;                /* az eddigi maximális hossz */
    char sor[MAXSOR];       /* az aktuális sor */
    char leghosszabb[MAXSOR];
    /* ide teszi a leghosszabb sort */

    max = 0;
    while ((hossz = getline(sor, MAXSOR)) > 0)
        if (hossz > max) {
            max = hossz;
            copy(leghosszabb, sor);
        }
    if (max > 0)            /* volt sor, nem EOF */
        printf("%s", leghosszabb);
    return 0;
}
/* getline: egy sort beolvas az s-be */
/* és visszaadja a hosszát */
int getline(char s[ ], int lim)
{
    int c, i;

    for (i = 0; i < lim-1 && (c = getchar()) != EOF
            && c != '\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}
```

```
/* copy: a "ba" helyre másol a "bol" helyről */
void copy(char ba[ ], char bol[ ])
{
    int i;

    i = 0;
    while ((ba[i] = bol[i]) != '\0')
        ++i;
}
```

A `getline` és `copy` függvényeket a program elején deklaráljuk és a programról feltételezzük, hogy egyetlen állományban van.

A `main` és a `getline` egy argumentumpáron keresztül tartja a kapcsolatot és a `getline` egy értékkel tér vissza. A `getline` argumentumait az

```
int getline(char s[ ], int lim)
```

sorban deklaráltuk, ami azt mondja, hogy az első argumentum (`s`) egy tömb, a második (`lim`) pedig egy egész változó. A deklarációban a tömb méretének megadásától eltekinthetünk. Az `s` tömb méretét a `getline` függvényben sem kell megadni, mivel azt a `main`-ben már beállítottuk. A `getline` függvény a `power`-hez hasonlóan tartalmaz egy `return` utasítást, amelyen keresztül egy értéket ad vissza a hívó programnak. A deklaráló sor jelzi, hogy a `getline` egész típusú értéket ad vissza. Mivel az alapfeltételezés szerint a visszatérési érték `int` típusú, így a deklarációból ez el is hagyható.

Néhány függvény a hívó programban felhasználható értékkel tér vissza, mások (mint pl. a `copy`) csak végrehajtanak egy feladatot és nem adnak vissza értéket. A `copy` függvény visszatérési típusa `void`, ami explicit módon azt jelzi, hogy nincs visszatérési érték.

A `getline` a `'\0'` karaktert (*nulla karaktert,* amelynek értéke nulla) helyezi a tömb végére, amivel a karaktersorozat (a beolvasott sor) végét jelzi. A C nyelv is ezt a módszert használja a szöveg végének jelzésére. Például a

```
"halló\n"
```

karakterlánc-állandó egy tömbként tárolódik el, amelynek egyes elemei a karakterlánc egyes karakterei és a végét a `'\0'` végjel mutatja az alábbiak szerint:

h	a	l	l	o	\n	\0

A `printf` függvényben szereplő `%s` formátummegadás egy ilyen formában megadott karaktersorozat kiírását jelzi. A `copy` függvény is kihasználja, hogy a kapott argumentumát a `'\0'` karakter zárja és ezt a karaktert át is másolja a kimenő argumentumába. (Mindez azt jelenti, hogy a `'\0'` karakter nem része a beolvasott szövegnek.)

Egyértelmű, hogy még egy ilyen kis programnál is adódnak tervezési problémák. Például felmerül a kérdés: mit csináljon a `main`, ha a beolvasott sor hosszabb a megadott korlátnál? A `getline` jól működik: ha megtelt a tömb, leáll, még akkor is, ha nem olvasott újsor-karaktert. A `getline`-tól kapott hossz és az utolsó karakter alapján a `main` eldöntheti, hogy a sor túl hosszú volt-e, és ezután tetszése szerint cselekedhet. A program rövidsége miatt ezt az esetet nem vettük figyelembe.

A `getline` függvény felhasználója nem tudhatja, hogy milyen hosszú a beolvasandó sor, ezért a `getline` ellenőrzi a túlcsordulást. A `copy` felhasználója viszont már

tudja (vagy megtudhatja) a karaktersorozat hosszát, ezért ott nem alkalmaztunk hibaellenőrzést.

1.16. gyakorlat. Módosítsuk a leghosszabb sort kiíró program `main` függvényét úgy, hogy helyesen adja meg a tetszőlegesen hosszú bemeneti sor méretét és annak szövegéből a lehető legtöbbet írja ki!

1.17. gyakorlat. Írjunk programot, ami kiírja az összes, 80 karakternél hosszabb bemeneti sort!

1.18. gyakorlat. Írjunk programot, ami eltávolítja a beolvasott sorok végéről a szóközöket és tabulátorokat, valamint törli a teljesen üres sorokat!

1.19. gyakorlat. Írjunk egy `reverse(s)` függvényt, ami megfordítja az `s` karaktersorozat karaktereit! Használjuk fel ezt a függvényt egy olyan programban, ami soronként megfordítja a beolvasott szöveget.

1.10. A változók érvényességi tartománya és a külső változók

A `main`-ben használt változók (pl. `leghosszabb`, `sor` stb.) a `main` saját, lokális változói. Mivel ezeket a `main`-ben deklaráltuk, így közvetlenül egyetlen más függvény sem férhet hozzájuk. Ez ugyanígy igaz a többi függvényre is, pl. a `getline` `i` változójának semmi köze a `copy` `i` változójához. A függvény lokális változói csak a függvény hívásakor jönnek létre és megsemmisülnek, amikor a függvény visszaadja a vezérlést a hívó programnak. Ezért az ilyen változókat (más nyelvek szóhasználatához igazodva) *automatikus* változóknak nevezzük. Ezentúl a lokális változókra való hivatkozáskor az automatikus megnevezést fogjuk használni. (A 4. fejezetben tárgyaljuk majd a `static` tárolási osztályú változókat, amelyek két függvényhívás közt is megtartják értéküket.)

Mivel az automatikus változók csak a függvény hívásakor léteznek, így a következő hívásig nem őrzik meg az értéküket és minden függvényhíváskor explicit módon értéket kell nekik adni. Ha erről megfeledkeznénk, akkor a tartalmuk határozatlan lesz.

Az automatikus változók mellett olyan változók is definiálhatók, amelyek az összes függvényre nézve *külsők,* azaz amelyekhez a nevükre hivatkozva bármely függvény hozzáférhet. (Ezek a változók nagyon hasonlítanak a FORTRAN nyelv COMMON változóihoz vagy a Pascal programok legkülső blokkjában deklarált változókhoz.) Mivel ezek a külső (*external*) változók az egész programra nézve globálisak, felhasználhatók a függvények közti adatcseréhez az argumentumlista helyett. A külső változók állandóan érvényben vannak, és függetlenül a függvények hívásától vagy a függvényből való visszatéréstől, megtartják az értéküket.

A külső változókat csak egyszer, az összes függvényen kívül kell *definiálni,* és ennek hatására tárolóhely rendelődik hozzájuk. A külső változókat minden olyan függvényben *deklarálni* kell, amely hozzájuk akar férni. Ez a deklaráció megadja ezen változók típusát a függvényben. A deklaráció történhet explicit módon, az `extern` utasítással vagy implicit módon, a programkörnyezet alapján. A külső változók használatának megvilágítására írjuk újra a leghosszabb sort kiíró programot úgy, hogy a `sor`, a `leghosszabb` és a `max` változók külső változók legyenek. Ez a függvényhívások, a deklarációk és mindhárom függvénytörzs módosítását igényli.

Az új program:

```
#include <stdio.h>

#define MAXSOR 1000 /* a beolvasott sor max. mérete */
```

```
int max;                    /* az eddigi maximális hossz */
char sor[MAXSOR];           /* az aktuális sor*/
char leghosszabb[MAXSOR];
/* ide teszi a leghosszabb sort */

int getline(void);
void copy(void);

/* a leghosszabb sor kiíratása - speciális változat */
main( )
{
    int hossz;              /* az aktuális sor hossza */
    extern int max;
    extern char leghosszabb[];

    max = 0;
    while ((hossz = getline( )) > 0)
        if (hossz > max) {
            max = hossz;
            copy( );
        }
    if (max > 0)            /* volt sor, nem EOF */
        printf("%s", leghosszabb);
    return 0;
}

/* getline: speciális változat */
int getline(void)
{
    int c, i;
    extern char sor[ ];

    for (i = 0; i < MAXSOR-1 && (c = getchar( )) != EOF
            && c != '\n'; ++i)
        sor[i] = c;
    if (c == '\n') {
        sor[i] = c;
        ++i;
    }
    sor[i] = '\0';
    return i;
}

/* copy: speciális változat */
void copy(void)
{
    int i;
    extern char sor[ ], leghosszabb[ ];

    i = 0;
    while ((leghosszabb[i] = sor[i]) != '\0')
        ++i;
}
```

A main, getline és copy függvények külső változóit a példaprogram első soraiban definiáltuk, amivel meghatároztuk a típusukat és lefoglaltuk számukra a tárolóhelyet. Szintaktikailag ezek a külső változódefiníciók ugyanolyanok, mint a lokális változók definíciói, de mivel a függvényeken kívül helyezkednek el, ezért külső változót írnak le. Egy függvény csak akkor használhat külső változókat, ha azok nevei már ismertek a számára. Ennek egyik módja, hogy egy extern deklarációt írunk a függvénybe. A deklaráció ugyanolyan, mint a korábbiak, csak az extern alapszó van előtte.

Bizonyos esetekben az extern deklaráció elmaradhat. Ha a külső változó definíciója a forrásállományban megelőzi a változó használatát valamelyik függvényben, akkor ebben a függvényben nem szükséges az extern deklaráció. Példaprogramunkban a main, getline és copy függvények extern deklarációi emiatt feleslegesek. Általános gyakorlat, hogy az összes külső változó definícióját a forrásállomány elejére teszik, és így az összes extern deklaráció elhagyható a programból.

Ha a program több forrásállományban van, és a külső változókat pl. a file1 állományban definiáljuk, és a file2, ill. file3 állományokban használjuk, akkor az extern deklarációra a változók előfordulásának megfelelően szükség van a file2 és file3 állományokban. Szokásos gyakorlat, hogy az összes külső változó extern deklarációját és a függvényeket egy önálló állományba (amit történelmi okokból fejnek vagy *header*-nek neveznek) gyűjtik és ezt az #include paranccsal az egyes forrásállományokhoz kapcsolják. Az állománynév utáni .h kiterjesztés ilyen header állományt jelöl. A standard könyvtár függvényei pl. az <stdio.h> -hoz hasonló header állományokban vannak deklarálva. A deklarációk részleteit a 4. fejezetben, a könyvtárral kapcsolatos ismereteket a 7. fejezetben és a B. Függelékben tekintjük át.

Mivel a getline és copy függvényeknek az új programváltozatban nincs argumentumuk, arra gondolhatnánk, hogy a forrásállomány elején a prototípusuk getline( ) és copy( ) alakú. De a régebbi C programokkal való kompatibilitás érdekében a szabvány az üres paraméterlistát régi stílusú függvénydeklarációnak tekinti és leállítja a paraméterlista ellenőrzését, ezért a ténylegesen üres paraméterlistában a void kulcsszónak kell szerepelnie. A kérdéssel a 4. fejezetben még foglalkozunk.

Megjegyezzük, hogy a külső változókkal foglalkozó részben nagy gonddal használtuk a *definiálás* és *deklarálás* fogalmakat. A definíció a program azon helye, ahol a változót (vagy függvényt) létrehoztuk vagy tárterületet rendeltünk hozzá. A deklaráció viszont olyan programrész, ahol csak leírjuk a változó tulajdonságait, de nem rendelünk hozzá tárterületet.

Hajlamosak vagyunk a program összes változóját külső változóként definiálni, mert így egyszerűsíthető a függvények közötti információcsere, rövidebbé válnak az argumentumlisták és a változók mindig a rendelkezésünkre állnak, amikor szükséges. De sajnos, a külső változók akkor is jelen vannak, ha nem akarjuk! Elég veszélyes dolog túlzottan a külső változókra támaszkodni, mert ez olyan programot eredményez, amelyben az adatkapcsolatok áttekinthetetlenek és a változók nem várt módon, sőt sokszor szándékunk ellenére megváltoznak, valamint a programot később nehéz módosítani. A leghosszabb sort kiíró program második változata ilyen szempontból rosszabb az elsőnél és tovább rontja a helyzetet, hogy a változók nevének rögzítésével a getline és a copy függvények elvesztették általános jellegüket.

Ebben a fejezetben áttekintettük a C nyelv legfontosabb elemeit. Ezekből az elemekből jelentős méretű, jól használható programok írhatók. Ennek érdekében javasoljuk az olvasónak, hogy most tartson egy kis szünetet a könyv olvasásában, és mielőtt tovább haladna, gondolja át az itt leírtakat, tanulmányozza a példaprogramokat és oldja meg a gyakorlatok feladatait. A következő gyakorlatokkal olyan programozási feladatokat kínálunk, amelyek a korábbiaknál bonyolultabbak.

1.20. gyakorlat. Írjunk `detab` néven programot, amely a beolvasott szövegben talált tabulátorkaraktereket annyi szóközzel helyettesíti, amennyi a következő tabulátorpozícióig hátravan! Tételezzük fel, hogy a tabulátorpozíciók adottak, pl. minden *n*-edik oszlopban. Az *n* értékét változóként vagy szimbolikus állandóként célszerű megadni?

1.21. gyakorlat. Írjunk programot `entab` néven, amely a beolvasott szövegben talált, szóközökből álló karaktersorozatot a minimális számú tabulátorkarakterrel és szóközökkel helyettesíti úgy, hogy a szövegben a távolság ne változzon! Használjuk ugyanazokat a tabulátorpozíciókat, mint a `detab` programban! Ha a következő tabulátorpozíció eléréséhez egyetlen szóköz vagy egyetlen tabulátor karakter is elegendő, akkor melyiket részesíti előnyben?

1.22. gyakorlat. Írjunk olyan programot, amely a hosszú bemeneti sorokat az *n*-edik oszlop előtt előforduló utolsó szóközkarakter után egy vagy több rövidebb sorba tördeli! Győződjünk meg arról, hogy a program nagyon hosszú sorok és az *n*-edik oszlop előtt sem szóközt, sem tabulátort nem tartalmazó sorok esetén egyaránt helyesen működik!

1.23. gyakorlat. Írjunk programot, ami egy C program szövegéből eltávolít minden megjegyzés szövegrészt! Ne feledkezzünk meg az idézőjelek közti karaktersorozatok és karakterállandók helyes kezeléséről! A C nyelvben a megjegyzés szövegek nem ágyazhatók egymásba.

1.24. gyakorlat. Írjunk programot, ami egy C program szövegét olyan alapvető szintaktikai hibák szempontjából ellenőrzi, mint a nem azonos számú kerek, szögletes és kapcsos kezdő és végzárójelek! Ne feledkezzünk meg az idézőjelekről, aposztrófokról, escape jelsorozatokról és megjegyzés szövegekről sem! Ezt a programot teljesen általános formában elég nehéz elkészíteni.

2. FEJEZET:

Típusok, operátorok és kifejezések

A változók és az állandók alkotják a programban feldolgozott alapvető adatobjektumokat. A program deklarációi felsorolják a felhasznált változókat, megadják a típusukat és néha még a kezdeti értéküket is. Az operátorok azt határozzák meg, hogy mit kell csinálni az adatokkal. A kifejezések a változókat és állandókat egy új érték előállítása érdekében kombinálják. Az objektum típusa meghatározza, hogy az objektum milyen értékeket vehet fel és milyen operátorok alkalmazhatók rá. Ebben a fejezetben ezekkel az alapelemekkel foglalkozunk.

Az ANSI szabvány csak kis változásokat hozott az alapvető adattípusok és kifejezések terén, és inkább csak bővítette a lehetőségeket. Újdonság az összes egész típusú mennyiségre bevezetett `signed` és `unsigned` forma, valamint az előjel nélküli (`unsigned`) állandó és a hexadecimális karakteres állandó bevezetése. Az új szabvány szerint a lebegőpontos műveleteket egyszeres pontossággal végzi a gép, és a nagyobb pontosság eléréséhez bevezették a `long double` adattípust. A karakterlánc-állandók a fordítás során összekapcsolhatók. A felsorolt változók a nyelv részévé váltak a rég bevált tulajdonságok formalizálásával. Deklarálhatóvá vált a `const` típusú objektum, ami ezek változatlanságát jelzi. Az aritmetikai adattípusok közti automatikus konverziós kényszerek bővítik az adattípusok választékát.

2.1. Változónevek

Bár az 1. fejezetben nem említettük, de van néhány megszorítás a változók és szimbolikus állandók neveit illetően. A nevek betűkből és számjegyekből állhatnak és az első karakterüknek betűnek kell lenni. Az aláhúzás-karakter (_) betűnek számít, és alkalmazásával sokszor javítható a hosszú változónevek olvashatósága. Változónevet ne kezdjünk aláhúzás-karakterrel, mivel a könyvtári eljárások gyakran használnak ilyen neveket. A nagy- és kisbetűk különböznek egymástól, így az `x` és `X` két különböző nevet jelent. A hagyományos C programozói gyakorlatban a változóneveket kisbetűvel írjuk és a szimbolikus állandókat csupa nagybetűvel.

A belső neveknek legalább az első 31 karaktere szignifikáns. A függvények és külső változók neveinek hossza 31-nél kisebb, mert ezek külső nevek, amelyeket az assemblerek és loaderek használnak a nyelvtől függetlenül (ezekre nem vonatkoznak a C nyelv szabályai). A szabvány külső nevek esetén csak 6 karakterig garantálja a megkülönböztethetőséget. A nyelv kulcsszavai (pl. `if`, `else`, `int`, `float` stb.) fenntartott szavak és nem lehetnek változónevek. A kulcsszavakat kisbetűvel kell írni.

Célszerű a programban olyan neveket választani, amelyek jelentenek valamit („beszélő nevek") és írásmódjuk nem zavaró. Érdemes a helyi változókhoz (különösen a ciklusváltozóhoz) rövid, a külső változókhoz hosszabb neveket választani.

2.2. Adattípusok és méretek

A C nyelv viszonylag kevés alapvető adattípust használ:

`char`	egyetlen bájt, a gépi karakterkészlet egy elemét tárolja
`int`	egész szám, mérete általában a befogadó számítógép egészek ábrázolásához használt mérete
`float`	egyszeres pontosságú lebegőpontos szám
`double`	kétszeres pontosságú lebegőpontos szám

Ezekhez az alapvető adattípusokhoz még néhány minősíthető specifikáció járulhat. Ilyen az egész típusokhoz használható short és long minősítő, pl.

```
short int sh;
long int counter;
```

Az ilyen deklarációkból az `int` típusjelzés elhagyható, és általában el is hagyják.

A `short` és `long` minősítők bevezetésével az volt a cél, hogy ahol szükséges, két különböző hosszúságú egész álljon a programozó rendelkezésére. Szokásos módon az int mérete megegyezik a használt számítógép alap szóméretével. A `short` típus általában 16 bites, a `long` pedig 32 bites, az `int` típus 16 vagy 32 bites. Minden fordítóprogram a saját hardverének megfelelően választja meg az adatok méretét, és csak annyi megszorítás van, hogy a `short` és `int` adattípus legalább 16, a long adattípus legalább 32 bites kell legyen, valamint, hogy a `short` nem lehet hosszabb az `int`-nél és az nem lehet hosszabb a `long`-nál.

A `signed` és `unsigned` minősítők a `char` és bármely egész adattípus esetén használhatók. Egy unsigned típusú szám mindig pozitív vagy nulla, és a modulo 2^n aritmetika szabályai szerint használható, ahol n az adott típus ábrázolásához rendelt bitek száma. Így pl. ha a `char` típust a gép 8 biten ábrázolja, akkor az `unsigned char` típus értéke 0 és 255 között, amíg a `signed char` típus értéke –128 és +127 között lehet (kettes komplemens kódú ábrázolás esetén). Az, hogy a `char` típusú adat `signed` vagy `unsigned`-e a géptől függ, de a nyomtatható karakterek mindig pozitívak.

A `long double` típus növelt pontosságú lebegőpontos számot jelöl. Az egészekhez hasonlóan a lebegőpontos számok mérete is gépfüggő. A `float`, `double` és `long double` típus egyszeres, kétszeres és háromszoros méretet jelenthet.

A `<limits.h>` és `<float.h>` szabványos header állományok szimbolikus állandói között mindezen méretűek megtalálhatók, és ezek leírása a géptől, ill. a fordítóprogramtól függő tulajdonságokkal együtt a B. Függelékben található.

2.1. gyakorlat. Írjunk programot, ami meghatározza a `signed` és `unsigned` minősítőjű `char`, `short`, `int` és `long` típusú változók nagyságát a szabványos header állományokból vett megfelelő értékek kiírásával és közvetlen számítással! A feladat nehezebb, ha kiszámítjuk a nagyságokat és tovább nehezíthető, ha a lebegőpontos számok nagyságát is meg akarjuk határozni.

2.3. Állandók

Az `1234` formában leírt egész állandó minden külön jelzés nélkül `int` típusú. Egy `long` típusú egész állandó leírásánál viszont a számot `l` (el) vagy `L` betűvel kell zárni,

pl. `123456789L`. Ez a szám túl nagy ahhoz, hogy `int` típusú legyen, ezért `long` lesz. Az előjel nélküli (`unsigned`) számokat az utánuk írt `u` vagy `U` betűvel jelöljük, és az `ul` vagy `UL` toldalék `unsigned long` típust ír elő.

A lebegőpontos állandók tizedespontot (pl. `123.4`) vagy kitevőt (pl. `1e-2`) vagy mindkettőt tartalmaznak, és alapértelmezésben `double` típusnak. A lebegőpontos állandó után írt `f` vagy `F` float, `l` vagy `L` long `double` típust jelöl.

Egy egész szám értéke nem csak decimálisan, hanem oktális vagy hexadecimális alakban is megadható. A szám elé nullát írva az egész típusú állandó oktális alakú és ha `0x` jelzést írunk a szám elé, akkor hexadecimális alakú. Például a decimális `31` oktális alakban `037` és hexadecimális alakban `0x1f` vagy `0X1F`. Az oktális vagy hexadecimális állandó után írt `L long` és `U unsigned` típust jelöl: pl. `0XFUL` egy decimálisan `15` értékű `unsigned long` típusú állandó.

A *karakterállandó* egy egész típusú adat, amit aposztrófok közé írt egyetlen karakterrel (pl. `'x'`) adunk meg. A karakterállandó értéke a karakter gépi karakterkészletbeni kódszáma. Például az ASCII karakterkészletben a `'0'` karakterállandó értéke `48`, ami semmiféle kapcsolatban nincs a `0` számértékkel. Ha a 48 kódérték helyett '0' karakterállandót írunk, akkor a program függetlenné válik a gépi karakterkészlettől és könnyebben olvasható. A karakterállandókat leggyakrabban más karakterekkel való összehasonlításra használjuk, de ugyanúgy részt vehetnek a numerikus műveletekben is, mint bármilyen egész adat.

Bizonyos karakterek a karakterállandókban vagy karaktersorozat-állandókban escape sorozattal adhatók meg, mint pl. a `\n`, ami az újsor-karaktert jelöli. Ezek az escape sorozatok leírva két karakternek látszanak, de valójában egyetlen karaktert jelölnek. Mindezeken kívül tetszőleges tartalmú, egy bájt méretű bitminta adható meg a

'*ooo*'

specifikációval, ahol *ooo* egy max. háromjegyű oktális szám számjegyeit jelöli (csak a `0...7` számjegyek megengedettek), vagy a

'*xhh*'

specifikációval, ahol *hh* egy max. kétjegyű hexadecimális szám jegyeit jelöli (a számjegyek `0...9`, `a...f` vagy `A...F` lehetnek). Így minden további nélkül írhatjuk, hogy

```
#define VTAB '\013'      /* a VTAB ASCII kódja */
#define BELL '\007'      /* a BELL ASCII kódja */
```

vagy hexadecimálisan

```
#define VTAB '\xb' /* a VTAB ASCII kódja */
#define BELL '\x7' /* a BELL ASCII kódja */
```

Az escape sorozatok teljes listája a következő:

\a	figyelmeztető jelzés (bell, csengő)
\b	visszalépés (backspace)
\f	lapdobás (formfeed)
\n	új sor (newline)
\r	kocsi vissza (carriage return)
\t	vízszintes tabulátor (horizontal tab, HTAB)

\v függőleges tabulátor (vertical tab, VTAB)
\\ fordított törtvonal (backslash)
\? kérdőjel
\' aposztróf
\" idézőjel
\ooo oktális szám
\xhh hexadecimális szám

A '\0' karakterállandó egy nulla értékű karaktert, az ún. null-karaktert jelöli. A programokban gyakran írunk '\0'-t a 0 helyett, hogy ezzel is kihangsúlyozzuk a kifejezésen belül az érték karakteres természetét, de természetesen mindkét alak számértéke nulla.

Az *állandó kifejezés* csak állandókat tartalmaz. Az ilyen kifejezések kiértékelése még a fordítás során megtörténik, ezért bárhol szerepelhetnek, ahol állandó használata megengedett. Ilyen állandó kifejezés pl. a

```
#define MAXSOR 1000
char sor[MAXSOR+1];
```

vagy a

```
#define UGRAS 1 /* pl. ugrás szökőévben */
int napok[31+28+UGRAS+31+30+31+30+31+31+30+31+30+31];
```

A *karaktersorozat-állandó (string-konstans)* nulla vagy több karakterből áll, amelyek idézőjelek között helyezkednek el. Ilyen pl. a

```
"Ez egy karaktersorozat"
```

vagy a

```
"" /* ez egy üres karaktersorozat */
```

Az idézőjelek nem részei a karaktersorozatnak, csak határolják azt. A karaktersorozat-állandókban ugyanazok az escape sorozatok használhatók, mint a karakterállandóknál, ezért a \" egy idézőjelet jelent (mivel az idézőjel határolja a karaktersorozatot, ezért annak belsejében idézőjel csak a \" escape sorozattal íratható ki). A karaktersorozat-állandók a fordítás során összekapcsolhatók (konkatenálhatók), így a

```
"Halló" " mindenki!"
```

egyenértékű a

```
„Halló mindenki!"
```

karaktersorozattal. Ez lehetőséget nyújt arra, hogy a forrásprogram túl hosszú sorait több, rövid sorra bontsuk.

Gyakorlatilag a karaktersorozat-állandó egy karakterből álló tömb. Egy karaktersorozat belső ábrázolásában a sorozatot a '\0' null-karakter zárja, így a tárolásukhoz szükséges tárolóhelyek száma csak eggyel több, mint a karakterek száma. Ez a belső ábrázolásmód azt eredményezi, hogy nincs korlátozva a karaktersorozat hossza, de

így a programoknak a teljes karaktersorozatot végig kell nézni ahhoz, hogy meghatározzák a tényleges hosszt. A standard könyvtár `strlen(s)` függvénye visszatéréskor megadja a karaktersorozat típusú `s` argumentumának a hosszát (a záró `'\0'` null-karaktert nem számítja bele a hosszba). A függvény általunk írt változata:

```
/* strlen: az s karaktersorozat hosszát adja */
int strlen(char s[ ])
{
    int i;

    i = 0;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

Az `strlen` és még több más karaktersorozat-kezelő függvény is a `<string.h>` szabványos headerben található.

Ügyeljünk arra, hogy megkülönböztessük a karakterállandót és az egyetlen karaktert tartalmazó karaktersorozatot: `'x'` nem azonos az `"x"`-szel! Az első egy egész mennyiség, amelynek számértéke az x betű gépi karakterkészletbeli kódja, amíg a második egy egyetlen karaktert (az *x* betűt) és a lezáró `'\0'` null-karaktert tartalmazó tömb.

Az állandók eddigiektől eltérő fajtája a *felsorolt állandó*. A felsorolt állandó egész értékek listájából áll, mint pl. az

```
enum boolean { NO, YES };
```

Az `enum` listájában az első név értéke 0, a másodiké 1 és így tovább, kivéve ha az értékeket explicit módon specifikáljuk. Ha a listában nem minden értéket specifikáltunk, akkor a nem specifikált elemek az utolsó specifikált elemtől kezdve folyamatosan a következő értéket kapják, mint ez az alábbik közül a második példában látható:

```
enum escapes
{ BELL = '\a', BACKSPACE = '\b', TAB = '\t',

NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };
enum honapok { JAN = 1, FEB, MAR, APR, MAJ,
               JUN, JUL, AUG, SZEP, OKT, NOV, DEC };
               /* így FEB = 2, MAR = 3 stb. */
```

A különböző felsorolt állandókban szereplő neveknek különbözniük kell, de egy felsoroláson belül az értékeknek nem kell különbözni.

A felsorolás egy szokásos módja annak, hogy a nevekhez állandó értéket rendeljünk, és így lényegében a `#define` utasítás helyettesítésére alkalmas, azzal az előnnyel, hogy a nevekhez rendelt értékek automatikusan generálhatók. Bár az `enum` típusú változókat deklarálhatjuk, a fordítóprogramok nem ellenőrzik, hogy az így eltárolt változóknak van-e érvényes értékük a felsorolásban. Mindazonáltal a felsorolt változók jellegéből adódik az ellenőrzés lehetősége, így gyakran jobban használhatók, mint a `#define` utasítással definiált változók. További előny, hogy egy megfelelő debugger program szimbolikus formában is képes kiírni a felsorolt változók értékét.

2.4. Deklarációk

A felhasználása előtt minden változót deklarálni kell, bár bizonyos deklarációk implicit módon, a programkörnyezet alapján is létrejöhetnek. A deklaráció egy típust határoz meg, és utána egy vagy több adott típusú változó felsorolása (listája) áll.

Ilyen deklaráció pl.

```
int also, felso, lepes;
char c, sor[1000];
```

A vátlozók a deklarációk közt tetszőleges módon szétoszthatók, pl. a fenti deklarációs listákkal teljesen egyenértékű az

```
int also;
int felso;
int lepes;
char c;
char sor[1000];
```

Az utóbbi forma több helyet igényel, viszont előnye, hogy az egyes deklarációkhoz kényelmesen fűzhetők megjegyzések és maga a deklaráció is egyszerűen módosítható.

A változók a deklaráció során kezdeti értéket is kaphatnak. Ha a nevet egyenlőségjel és egy kifejezés követi, akkor a kifejezés értéke lesz a kezdeti érték, mint pl. a következő deklarációban:

```
char esc = '\\';
int i = 0;
int hatar = MAXSOR+1;
float eps = 1.0e-5;
```

Ha a kérdéses változó nem automatikus, akkor a kezdeti értékadás csak egyszer, a program végrehajtásának kezdetén történik meg és a kezdő értéket megadó kifejezésnek állandó kifejezésnek kell lenni. Az explicit módon inicializált automatikus változók esetén a kezdeti értékadás minden alkalommal létrejön, amikor a vezérlés átkerül a végrehajtandó függvényhez vagy blokkhoz, és a kezdeti értéket megadó kifejezés tetszőleges lehet. Alapfeltételezés szerint a külső és statikus változók nulla kezdeti értéket kapnak. Az automatikus változók kezdeti értéke határozatlan, hacsak explicit módon nem kapnak értéket.

Bármely változó deklarációjában alkalmazható a `const` minősítő, ami azt jelzi, hogy a változó értékét nem fogjuk megváltoztatni. Ilyen deklaráció pl.

```
const double e = 2.71828182845905;
const char uzenet[ ] = "figyelem: ";
```

A `const` minősítésű deklaráció tömb argumentumok esetén is használható, jelezve hogy a függvény nem változtatja meg a tömböt. Pl.:

```
int strlen(const char[ ]);
```

Amennyiben a program a `const` minősítésű változó értékének megváltoztatására mégis kísérletet tesz, akkor a hatás a használt számítógéptől és rendszertől függ.

2.5. Aritmetikai operátorok

A C nyelv kétoperandusú aritmetikai operátorai a `+`, `-`, `*` és `/`, valamint a `%` modulus operátor. Az egészek osztásakor a törtrészt a rendszer levágja, ezért van szükség a `%` modulus operátorra. Az

```
x % y
```

kifejezés az `x/y` egészosztás (egész) maradékát adja, és értéke nulla, ha `x` osztható `y`-nal. Például egy adott év szökőév, ha az évszáma osztható néggyel és nem osztható százzal, kivéve a 400-zal osztható évszámokat, amik szintén szökőévek. Ezért annak eldöntése, hogy egy adott év szökőév-e vagy sem, az alábbi programrészlettel lehetséges:

```
if ((ev % 4 == 0 && ev % 100 != 0) || ev % 400 == 0)
   printf("%d szökőév.\n", ev);
else
   printf("%d nem szökőév.\n", ev);
```

A `%` operátor nem alkalmazható `float` és `double` típusú adatokra. Negatív operandusok esetén az egészosztás hányadosának csonkítása, valamint a modulus előjele gépfüggő, és ugyanez igaz az esetlegesen előforduló túlcsordulásra és alácsordulásra is.

Az egyoperandusú (unáris) + és – operátorok precedenciája a legmagasabb. A kétoperandusú + és – operátorok precedenciája kisebb a `*`, `/` és `%` precedenciájánál. Az aritmetikai operátorok mindig balról jobbra haladva hajtódnak végre (a precedencia figyelembevételével).

A fejezet végén lévő 2.1 táblázatban összefoglaltuk az összes operátor precedenciáját és asszociativitását.

2.6. Relációs és logikai operátorok

A C nyelv relációs operátorai: `>`, `>=` , `<`, `<=`. Ez a sorrend egyben a precedenciájuk sorrendje is. Ezeknél eggyel alacsonyabb precedenciájúak az egyenlőség operátorok: `==`, `!=`.

A relációs operátorok precedenciája kisebb, mint az aritmetikai operátoroké, így pl. egy olyan kifejezés, mint `i < hatar-1` úgy értékelődik ki, mint ha `i < (hatar-1)` formában írtuk volna (amint ez elvárható).

Sokkal érdekesebb az `&&` és `||` logikai operátorok kérdése! Az `&&` és `||` operátorokkal összekapcsolt kifejezések kiértékelése balról jobbra történik, és a kiértékelés azonnal félbeszakad, ha az eredmény igaz vagy hamis volta ismertté válik. A legtöbb C nyelvű program kihasználja ezeket a tulajdonságokat. Így pl. az 1. fejezetben írt `getline` függvény ciklusszervezése is ennek alapján működik:

```
for (i = 0; i<lim-1 && (c = getchar( )) != '\n'
       && c != EOF; ++i)
   s[i] = c;
```

Az új karakter beolvasása előtt meg kell vizsgálni, hogy van-e hely számára az `s` tömbben, így először az `i<lim-1` ellenőrzést *kell* végrehajtani. Ha ennek az ered-

ménye hamis, akkor a program már nem is megy tovább a következő karakter beolvasására. Ugyancsak nem volna szerencsés, ha a c karaktert az állomány vége feltételre vizsgálnánk a getchar hívása előtt, ezért a hívásnak és az értékadásnak meg kell előzni a c karakter vizsgálatát.

Az && precedenciája nagyobb, mint a || precedenciája, és mindkét operátor alacsonyabb precedenciájú, mint a relációs és egyenlőség operátorok, így az

```
i<lim-1 && (c = getchar( )) != '\n' && c != EOF
```

kifejezés nem tartalmaz felesleges zárójeleket. De mivel a != precedenciája nagyobb, mint az értékadásé, ezért zárójelezés szükséges:

```
(c = getchar( )) != '\n'
```

Ezzel elérjük, hogy először az értékadás történjen meg, és csak ezután hasonlítsuk össze a c értékét az '\n' karakterrel.

Egy relációs vagy logikai kifejezés számértéke definíció szerint 0, ha a kifejezés hamis, és 1, ha igaz.

A ! unáris (egyoperandusú) negáló operátor a nem nulla (igaz) operandust 0 értékűvé (hamissá), a 0 értékű (hamis) operandust 1 értékűvé (igazzá) alakítja. A ! operátort gyakran használjuk olyan szerkezetekben, mint pl. az

```
if (!igaz)
```

az

```
if (igaz == 0)
```

kifejezés helyett. Nehéz általános esetben megmondani, hogy melyik változat a jobb. A !igaz szerkezet általában jól olvasható („nem igaz"), de bonyolultabb esetben nehezen érthető.

2.2. gyakorlat. Írjunk az előző for ciklussal egyenértékű ciklust, ami nem használja az && vagy || operátorokat!

2.7. Típuskonverziók

Ha egy operátor operandusai különböző típusúak, akkor a művelet végrehajtása előtt azokat egy közös típusra kell hozni. Az átalakításra néhány szabály vonatkozik. Általában az automatikus konverzió csak akkor jön létre, ha egy „keskenyebb" operandust egy „szélesebb" operandussá kell alakítani, mivel így biztosan nem vész el információ. Ilyen konverzió jön létre pl. az f+i alakú kifejezésekben, ahol az i egész lebegőpontossá alakul az f lebegőpontos szám miatt. Az értelmetlen kifejezések, mint pl. float típusú adat indexként való használata, nem megengedettek. Ha egy hosszabb egész típust egy rövidebbhez, vagy egy lebegőpontos típust egy egészhez rendelünk, akkor információ veszhet el, ezért figyelmeztető jelzést kapunk, de maga a művelet nem tilos.

A char típusú adatok kis egész számok, ezért az összes aritmetikai műveletben szabadon használhatók, ami nagymértékben egyszerűsíti és rugalmassá teszi a karakterkezelést. Erre jó példa az atoi függvény, amely a számjegyekből álló karaktersorozatot a megfelelő értékű számmá alakítja. A függvény egyszerűsített változata:

```
/* atoi: az s karaktersorozat egész számmá alakítása */
int atoi(char s[ ])
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

Amint azt az 1. fejezetben már elmondtuk, az s[i] – '0' kifejezés megadja az s[i]-ben tárolt karakter számértékét, mivel a '0', '1' stb. folyamatosan növekvő sorozatot alkot.

A char típus int típussá alakítására egy másik példa a lower függvény, amely egy (kizárólag ASCII karakterkészletbeli) karaktert alakít kisbetűvé. Ha a karakter nem nagybetű, akkor a függvény változatlan formában adja vissza.

```
/* lower: a c ASCII karakter kisbetűssé alakítása */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

A függvény csak ASCII karakterkészlet esetén használható, mivel kihasználja, hogy a nagy- és kisbetűk számértéke (kódja) rögzített távolságra van egymástól, és a nagy-, ill. kisbetűs ábécék folyamatosak (azaz A és Z, ill. a és z között a betűkön kívül nincs más karakter). Ez az utóbbi kitétel nem igaz az EBCDIC karakterkészletre, így a lower függvényünk nem működne helyesen (nem csak betűket alakítana át).

A B. függelékben leírt <ctype.h> standard header állományban számos függvény van, amelyek a karakterkészlettől független karakter-ellenőrzésre és -átalakításra használhatók. Például a tolower(c) függvény a c karakter kisbetűs értékével tér vissza, ha az nagybetűs karakter volt, így általánosabb formában helyettesítheti a lower függvényünket.

Az atoi függvényben használt

```
c >= '0' && c <= '9'
```

vizsgálat is helyettesíthető az

```
isdigit(c);
```

könyvtári függvénnyel. A továbbiakban a <ctype.h> standard header függvényeit használni fogjuk az egyes programokban.

A karakterek egésszé alakításával kapcsolatban szólnunk kell még a C nyelv egy további finomságáról. A nyelv nem határozza meg, hogy a char típusú változó előjeles vagy előjel nélküli mennyiség-e. Vajon char típusú adat int típusúvá alakításakor létrejöhet-e negatív szám? A válasz gépről gépre változik, a felépítéstől függően. Néhány szá-

mítógépnél azok a karakterek, amelyek bal szélső (legnagyobb helyiértékű) bitje 1, negatív egész számmá konvertálódnak (előjel-kiterjesztés!). Más gépeken az átalakítás során keletkező `int` típusú adat balról nullákkal töltődik fel, ezért mindig pozitív marad.

A C nyelv definíciója garantálja, hogy a gép szabványos karakterkészletében lévő egyetlen nyomtatható karakter sem lesz negatív, így ezek a karakterek egy kifejezésben mindig pozitív mennyiségként szerpelnek. Ez természetesen nem igaz a karakteres változóban tárolt tetszőleges bitmintákra, amelyek néhány gépen negatív, más gépeken pozitív értékűek lehetnek. Az egyértelműség és hordozhatóság érdekében célszerű a `signed` vagy `unsigned` minősítőt használni, ha nem karakteres adatot tárolunk `char` típusú változóban.

Az `&&` és `||` logikai műveletekkel összekapcsolt `i > j` alakú relációs kifejezések és logikai kifejezések eredményének értékét úgy definiáljuk, hogy az 1, ha az eredmény igaz, és 0, ha hamis. Így a

```
d = c >= '0' && c <= '9'
```

értékadás a `d` változóhoz 1 értéket rendel, ha `c` számjegy, és 0 értéket, ha nem. Másrészről az `isdigit`hez hasonló függvények igaz esetben tetszőleges nem nulla értékkel térnek vissza. Az eltérő értelmezések okozta hibák kiküszöbölése érdekében az `if, while, for` stb. utasítások vizsgáló részében az „igaz" ugyanazt jelenti, mint a „nem nulla", ezért a kétféle értékadás hatása között nincs különbség.

Az implicit aritmetikai típuskonverziók sokkal inkább a várakozásnak megfelelően működnek. Ha egy kétoperandusú operátor (pl. + vagy *) operandusai különböző típusúak, akkor az „alacsonyabb" típus „magasabb" típussá alakul (*„előlép"*) a művelet végrehajtása előtt, és az eredmény a „magasabb" típusnak megfelelő lesz. A pontos konverziós szabályokat az A. Függelék 6. pontjában ismertetjük, addig is néhány egyszerű szabályt adunk. Ha az operandusok nem `unsigned` minősítésűek, akkor:

> Ha valamelyik operandus `long double` típusú, akkor a másik is `long double` típusúvá konvertálódik;
> különben, ha valamelyik operandus `double` típusú, akkor a másik is `double` típusúvá konvertálódik;
> különben, ha valamelyik operandus `float` típusú, akkor a másik is `float` típusúvá konvertálódik;
> különben a `char` és `short` típusú operandus `int` típusúvá konvertálódik;
> végül, ha valamelyik operandus `long` típusú, akkor a másik is `long` típusúvá konvertálódik.

Megjegyezzük, hogy egy kifejezésben a `float` típusú operandusok nem automatikusan alakulnak át `double` típusúvá, ami a C nyelv eredeti definíciójához képest eltérést jelent. Általában a matematikai függvények (mint a `<math.h>` standard header függvényei is) kétszeres pontosságú lebegőpontos adatokat használnak. A `float` típus használatának fő oka, hogy nagy tömbök esetén tárterület takarítható meg, vagy ritkábban, hogy a végrehajtási idő csökken, mivel a kétszeres pontosságú aritmetika viszonylag lassú.

A konverziós szabályok `unsigned` minősítésű operandusok esetén sokkal bonyolultabbak. A problémát az előjeles és előjel nélküli számok összehasonlítása jelenti, mivel az értékek a géptől és a különböző egész típusok méretétől függenek. Például tegyük fel, hogy az `int` 16 bites, a `long` 32 bites. Ekkor `-1L < 1U`, mert `1U` (ami egy `int` típus) „előlép" `signed long` típussá. Másrészt viszont `-1L > 1UL`, mert a `-1L unsigned long` típussá „lép elő", amivel nagy pozitív számmá válik.

Az értékadás is típuskonverzióval jár: a jobb oldal értéke a bal oldal típusának megfelelő típusúvá alakul és ez lesz az eredmény típusa is.

A karakterek mindig egésszé alakulnak (előjel-kiterjesztéssel vagy anélkül, ahogy erről már volt szó).

A hosszabb egészek rövidebb vagy `char` típusúvá alakulásakor a magasabb helyiértékű bitek elvesznek. Így az

```
int i;
char c;

i = c;
c = i;
```

műveletsorban a `c` értéke változatlan marad. Ez mindig igaz, függetlenül attól, hogy van-e előjel-kiterjesztés vagy sem.

Az értékadás sorrendjének megfordítása információvesztéshez vezethet. Ha az `x` `float` és az `i` `int` típusú, akkor az `x = i` és `i = x` értékadásoknál mindkét esetben létrejön a típuskonverzió. A `float` `int` típussá alakulása a törtrész elvesztésével jár. Ha `double` típust alakítunk `float` típussá, akkor az új érték a rendszertől függően kerekítéssel vagy levágással keletkezik.

Mivel függvényhíváskor az argumentum kifejezés is lehet, az argumentum átadásakor is létrejöhet típuskonverzió. Függvényprototípus hiányában a `char` és `short` típusok `int` típussá, a `float` típus `double` típussá alakul. Ezért a függvények argumentumait `int` és `double` típusúnak deklaráltuk akkor is, ha a függvényt `char` vagy `float` típusú argumentummal hívtuk.

Végül megjegyezzük, hogy tetszőleges kifejezésben kikényszeríthetjük az explicit típuskonverziót a rögzítő (`cast`) unáris típusmódosító operátorral. Az így kialakított szerkezet:

(*típusnév*) *kifejezés*

alakú, és hatására a *kifejezés* a típusnévvel megadott típusúvá konvertálódik, a korábban elmondott szabályok szerint. A kényszerített típusmódosítás alapvetően úgy működik, mintha a *kifejezés* értékét az előírt típusú változóhoz rendelnénk egy értékadással, majd ezután ezt a változót használnánk a kifejezés helyett. Például a `sqrt` könyvtári függvény `double` típusú argumentumot vár, és értelmetlen eredményt ad, ha véletlenül valamilyen más típusú argumentumot kap. (Az `sqrt` függvény a `<math.h>` standard headerben található.) Így, ha az `n` változó egész típusú, akkor az `sqrt` függvény

```
sqrt((double) n)
```

formában használható, mivel ez a paraméterátadás előtt `double` típusúvá alakítja `n` értékét. Meg kell jegyeznünk, hogy bár a kényszerített típusmódosítás létrehozza `n` megfelelő típusú értékét, de magát az `n` változót nem változtatja meg. A kényszerített típusmódosítás operátora a többi unáris operátorral azonos precedenciájú, mint az a fejezet végén lévő összefoglaló táblázatból látszik.

Ha az argumentumot egy függvényprototípusban deklaráljuk (ami a szokásos megoldás), akkor a deklaráció bármilyen más, a függvény hívásakor felhasznált argumentumra rákényszeríti a megadott típust. Így pl. az `sqrt` prototípusát

```
double sqrt(double);
```

formában megadva megengedett a

```
gyok2 = sqrt(2)
```

alakú hívás, mivel a 2 automatikusan `double` típusú `2.0` értékké alakul, minden kényszerített típusmódosítás nélkül.

A standard könyvtár tartalmaz egy hordozható pszeudovéletlenszám-generátort, valamint annak kezdőértékét beállító függvényt. A két függvény egyszerűsített programja jó példa a kényszerített típusmódosításra. A program:

```
unsigned long int next = 1;

/* rand: egy 0 és 32767 közti pszeudo-véletlen számot */
/* ad vissza */
int rand(void);
{
    next = next * 1103515245 + 12345;
    return (unsigned int) (next/65536) % 32768;
}

/* srand: a rand kezdőértékének beállítása */
void srand(unsigned int alap)
{
    next = alap;
}
```

2.3. gyakorlat. Írjunk `htoi(s)` néven függvényt, amely egy hexadecimális számjegyekből álló karaktersorozatot (beleértve a `0x` vagy `0X` karaktersorozatot is) a megfelelő egész számmá alakít! A megengedett számjegyek `0...9` és `a...f` vagy `A...F`.

2.8. Inkrementáló és dekrementáló operátorok

A C nyelv két szokatlan operátort használ a változók inkrementálására (eggyel való növelésére) és dekrementálására (eggyel való csökkentésére). A `++` inkrementáló operátor egyet ad az operandushoz, a `--` dekrementáló operátor pedig egyet kivon belőle. A `++` operátort gyakran használjuk változók növelésére, pl. az

```
if (c == '\n')
    ++nl;
```

programrészletben. A `++` és `--` szokatlan vonatkozása, hogy prefix formában (a változó előtt elhelyezve, pl. `++n`) és postfix formában (a változó után elhelyezve, pl. `n++`) egyaránt létezik. A kétféle változat egyaránt növeli (vagy csökkenti) a változót, de a `++n` a felhasználás *előtt,* az `n++` pedig *utána* növeli az `n` értékét (a `--` operátor hasonlóan működik). Ebből következően minden olyan esetben, amikor a változó értékét is felhasználjuk (nem csak a növelésre vagy csökkentésre, azaz számlálásra van szükség), a `++n` és az `n++` különbözik. Ha pl. `n` értéke `5`, akkor

```
x = n++;
```

hatására x értéke 5 lesz, amíg az

```
x = ++n;
```

hatására x értéke 6 lesz. Természetesen n értéke mindkét esetben 6 lesz. Az inkrementáló vagy dekrementáló operátorok csak változókra alkalmazhatók, az (i + j)++ formájú kifejezések tilosak.

Olyan esetekben, amikor az értékre nincs szükség, csak a növelésre, mint pl. az alábbi példában:

```
if (c == '\n')
    nl++;
```

a prefix és postfix forma egyenértékű. Van viszont olyan feladat, amikor csak az egyiket vagy a másikat választhatjuk. Példaként vizsgáljuk meg a squeeze(s, c) függvényt, amely az s karaktersorozatból eltávolítja az összes c karaktert.

```
/* squeeze: az s-ből törli az összes c-t */
void squeeze(char s[ ], int c)
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

A függvény minden alkalommal, amikor c-től különböző karaktert talál, az i-edik pozícióból átmásolja azt a j-edik (aktuális) pozícióba, és csak ezután inkrementálja j értékét, hogy kijelölje a következő karakter helyét. Ez teljesen egyenértékű a következővel:

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

A másik példát az 1. fejezetben ismertetett getline függvényből vettük:

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

Ez a szerkezet helyettesíthető a sokkal tömörebb

```
if (c == '\n')
    s[++i] = c;
```

szerkezettel. A harmadik példánk az strcat(s, t) függvény, amely a t karaktersorozatot az s karaktersorozat végéhez illeszti (konkatenálja). Az strcat feltételezi, hogy az s karaktersorozat elegendően hosszú az összekapcsolt karaktersorozat befogadásához. Az itt közölt példaprogramban az strcat visszatéréskor nem ad értéket,

a standard könyvtári változata viszont egy mutatót ad vissza, ami kijelöli az eredményül kapott karaktersorozat helyét.

```
/* strcat: a t karaktersorozatot a s karaktersorozat végé-
hez kapcsolja, s elegendően hosszú */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0') /* megkeresi s végét */
        i++;
    while ((s[i++] = t[j++]) != '\0')
        ;       /* átmásolja t-t */
}
```

A program `t` minden egyes karakterének `s`-be másolása után a postfix ++ operátorral növeli `i` és `j` értékét, így azok a ciklusmag következő végrehajtásakor már a következő helyet jelölik ki.

2.4. gyakorlat. Írjuk meg a `squeeze(s1, s2)` olyan változatát, amely az `s1` karaktersorozatból töröl minden karaktert, ami az `s2` karaktersorozatban megtalálható!

2.5. gyakorlat. Írjunk `any(s1, s2)` néven függvényt, amely visszatérési értékként megadja az `s1` karaktersorozat azon legelső helyét, ahol az `s2` karaktersorozat bármelyik karaktere előfordul! A függvény visszatérési értéke legyen `-1`, ha `s1` egyetlen `s2`-beli karaktert sem tartalmaz. (Az `strbrk` standard könyvtári függvény ugyanezt teszi, csak visszatérési értékként az adott helyet kijelölő mutatót adja.)

2.9. Bitenkénti logikai operátorok

A C nyelvben hat operátor van a bitenkénti műveletekre. Ezek az operátorok csak egész típusú adatokra, azaz `char`, `short`, `int` és `long` típusokra használhatók, akár előjeles, akár előjel nélküli változatban. Az egyes operátorok és értelmezésük a következő:

`&`	bitenkénti ÉS-kapcsolat
`|`	bitenkénti megengedő (inkluzív) VAGY-kapcsolat
`^`	bitenkénti kizáró (exkluzív) VAGY-kapcsolat
`<<`	balra léptetés
`>>`	jobbra léptetés
`~`	egyes komplemens képzés (unáris)

A bitenkénti ÉS operátort gyakran valamilyen bitminta kimaszkolására használják, pl. az

```
n = n & 0177;
```

művelet az `n` bináris értékében az alsó hét bit kivételével minden bitet nulláz.

A bitenkénti VAGY operátort a bitek beállítására használják, pl. az

```
x = x | BEALL;
```

művelet x minden olyan bitjét 1-re állítja, amely a BEALL-ban is 1 volt (függetlenül attól, hogy korábban milyen értékű volt).

A kizáró VAGY operátor minden olyan helyen 1-et ad, ahol a két operandus bitjei különböztek, és 0-t, ha megegyeztek.

Az & és | bitenkénti, valamint az && és || logikai operátorok közti legfontosabb különbség, hogy az utóbbiak az igazságtábla szerint, balról jobbra haladva végzik a kiértékelést, míg az előzőek egy lépésben, bitenként. Például, ha x értéke 1 és y értéke 2, akkor x & y értéke nulla, viszont x && y értéke egy.

A << és >> léptető operátorok a bal oldalukon lévő operandust a jobb oldalukon lévő pozitív értéknek megfelelő számú bittel jobbra vagy balra léptetik. Így pl. az x << 2 művelet x értékét két hellyel balra lépteti, a jobb szélen keletkező két üres bináris helyet nullával tölti fel. A művelet megfelel a néggyel való szorzásnak. Egy előjel nélküli mennyiség jobbra léptetésekor a bal szélen felszabaduló bitek mindig nullával töltődnek fel. Előjeles számok jobbra léptetésekor a felszabaduló bitek az előjel bittel (aritmetikai léptetés), ill. néhány számítógép esetén nullával töltődnek fel (logikai léptetés).

A ~ unáris (egyoperandusú) operátor egy egész egyes komplemensét állítja elő, ami azt jelenti, hogy a szám minden 1 értékű bitjét 0-ra, minden 0 értékű bitjét 1-re állítja. Például az

```
x = x & ~077;
```

művelet hatására x utolsó hat bitje nulla értékű lesz. Vegyük észre, hogy az x ~ 077 eredménye független az x gépi ábrázolásához használt szó hosszától, és így 16 bites szóhossz esetén előnyösebb a vele egyenértékű (de csak akkor!) x ~ 0177700 művelet. A hordozható forma semmiféle többletbefektetést nem igényel, mivel az ~ 077 egy állandó kifejezés, aminek kiértékelése a fordítás során történik.

Néhány bitenkénti operátor működésének bemutatására írjunk getbits(x, p, n) néven függvényt, amely az x p-edik hellyel kezdődő n bites részét adja vissza, jobbra igazítva. Tételezzük fel, hogy a 0. bitpozíció a jobb szélső, valamint azt, hogy n és p értelmes pozitív mennyiségek. Például a getbits(x, 4, 3) x 4., 3. és 2. pozícióján álló hárombites, jobbra igazított számmal tér vissza. A függvény:

```
/* getbits: a p-edik pozíciótól kezdődő n bitet adja */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

az x >> (p+1-n) kifejezés a kiválasztott mezőt a szó jobb szélére mozgatja. A ~0 a teljes szóhosszon csupa 1 értékű bitet jelent, amit a ~0 << n művelettel n lépéssel balra tolunk. Ennek hatására a szó n számú jobb szélső bitje nullával töltődik fel, és ebből komplementálással alakul ki az n darab jobb szélső bitet kiválasztó (csupa egyesből álló) maszk.

2.6. gyakorlat. Írjuk meg a setbits(x, p, n, y) függvényt, amely egy olyan x értékkel tér vissza, amit úgy kap, hogy az x p-edik pozíciótól jobbra eső n bitje helyébe bemásolja y jobb szélső n bitjét, a többi bitet változatlanul hagyva!

2.7. gyakorlat. Írjunk egy invert(x, p, n) függvényt, amely az x p-edik pozíciótól kezdődő n bitjét invertálja (az 1-eseket 0-ra, a 0-kat 1-esekre változtatja), a többi bitet pedig változatlanul hagyja!

2.8. gyakorlat. Írjunk egy rightrot(x, n) függvényt, ami n bittel jobbra rotálja

az x egész mennyiséget! Jobbra rotálásnál a jobb szélen kilépő bitek a bal szélen visszakerülnek a szóba.

2.10. Értékadó operátorok és kifejezések

Az olyan kifejezéseket, mint

```
i = i + 2
```

amelyekben a bal oldalon lévő változó ismét megjelenik a jobb oldalon, az

```
i += 2
```

tömörebb formában is írhatjuk. Az új formában szereplő += jelkombinációt értékadó operátornak nevezik. A legtöbb kétoperandusú operátor (pl., mint a +, amelynek bal és jobb oldalán is operandus áll) szerepelhet az értékadó operátorban, amelynek általános alakja *op*=, ahol *op* a

```
+ - * / % << >> & ^ |
```

((c rész alatt operátorok egyike lehet. Ha *k1* és *k2* két kifejezés, akkor a

k1 op= k2

egyenértékű a

k1 = (*k1*) *op* (*k2*)

alakkal. A tömörebb forma előnye, hogy a gép a *k1* kifejezést csak egyszer számolja ki. Összetett kifejezések esetén ügyeljünk a zárójelekre. Például az

```
x *= y + 1
```

alak az

```
x = x * (y + 1)
```

kifejezésnek felel meg, és nem az

```
x = x * y + 1
```

kifejezésnek. Az elmondottak megvilágítására nézzük a `bitcount` függvényt, amely megszámolja az egész típusú argumentumában lévő 1 értékű biteket.

```
/* bitcount: x 1 értékű bitjeinek száma */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
```

```
        if (x & 01)
            b++;
    return b;
}
```

A függvény `x` argumentumát `unsigned` típusúnak deklaráltuk, hogy a jobbra léptetés során a felszabaduló helyekre garantáltan 0 értékű bitek lépjenek be, függetlenül a géptől, amelyen a program fut.

A tömörségen túl az értékadó operátorok alkalmazásának további előnye, hogy jobban megfelel az emberi gondolkodásnak. Általában azt mondjuk, hogy „adj kettőt `i`-hez" vagy „növeld `i`-t kettővel", ahelyett, hogy azt mondanánk „vedd `i`-t, adj hozzá kettőt, majd tedd az eredményt vissza az `i`-be". Emiatt az `i += 2` kifejezés előnyösebb, mint az `i = i+2`. Bonyolult kifejezéseknél az értékadó operátor használatának még az is előnye, hogy érthetőbbé teszi a programot.
Például egy

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

alakú kifejezésben nem kell nehézkes módon ellenőrizni, hogy a bal és jobb oldali kifejezések tényleg megegyeznek-e, ill. ha nem, akkor miért nem. Az értékadó operátorok a fordítóprogramot is segítik a hatékony kód előállításában.

Mint korábban már láttuk, az értékadó utasításnak értéke van és előfordulhat kifejezésben, amire a legegyszerűbb példa a

```
while ((c = getchar()) != EOF)
    ...
```

utasítássor. Ennek mintájára más értékadó operátort (`+=`, `-=` stb.) tartalmazó értékadások is szerepelhetnek kifejezésekben, bár ez nem túl gyakori.

Minden ilyen kifejezésben az értékadó kifejezés típusa megegyezik a bal oldali operandus típusával, és értéke az értékadás utáni érték.

2.9. gyakorlat. A kettes komplemens kódú aritmetikában az `x &= (x-1)` kifejezés törli `x` jobb szélső bitjét. Magyarázzuk meg, miért! Ezt kihasználva írjunk egy gyorsabb `bitcount` változatot!

2.11. Feltételes kifejezések

Az

```
    if (a > b)
        z = a;
    else
        z = b;
```

programrészlet hatására `z` az `a` és `b` értékek közül a nagyobbikat veszi fel. Ilyen és hasonló szerkezetek a C nyelv háromoperandusú `?:` *feltételes kifejezés* operandusával egyszerűbben is leírhatók. Az operátor általános formája kifejezésekben:

kif1 ? *kif2* : *kif3*

A szerkezet úgy működik, hogy először kiértékeli a *kif1* kifejezést. Ha ennek értéke

nem nulla (igaz), akkor a *kif2* kiértékelése következik, egyébként pedig a *kif3* kiértékelése. A program *kif2* és *kif3* közül csak az egyiket értékeli ki (a *kif1* értékétől függően) és ez lesz a feltételes kifejezés értéke. Így z beállítása az a és b közül a nagyobbik értékének megfelelően a

```
z = (a > b) ? a : b; /* z = max(a, b) */
```

utasítással történhet.

Meg kell jegyeznünk, hogy a feltételes kifejezés egy valódi kifejezés, tehát ugyanúgy használható, mint bármilyen más kifejezés. Ha *kif2* és *kif3* eltérő típusúak, akkor az eredmény típusát a korábban tárgyalt konverziós szabályok szerint lehet meghatározni. Például, ha `f float` típusú és `n int`, akkor az

```
(n > 0) ? f : n
```

kifejezés float típusú lesz, függetlenül attól, hogy n pozitív-e vagy sem.

A feltételes kifejezésben az első kifejezést nem szükséges zárójelbe tenni, mivel a `?:` operátor precedenciája nagyon alacsony (csak eggyel nagyobb, mint az értékadásé). Másrészt zárójelezéssel a kifejezés feltételrésze egyértelművé tehető.

A feltételes kifejezések használata tömör és világos kódot eredményez. Ezt jól mutatja az alábbi programrészlet, amely egy tömb n elemét nyomtatja ki úgy, hogy soronként tíz értéket ír, azokat egy szóközzel választja el és a sort újsor-karakterrel zárja.

```
for (i = 0; i<n; i++)
   printf("%6d%c", a[i], (i%10==9 || i==n-1) ?
      '\n' : ' ' );
```

A programrész minden tizedik sor és n-edik elem után egy újsor-karaktert ír ki, minden más esetben pedig a számot szóköz követi. A példa elég trükkös, de mindenképpen sokkal tömörebb, mint az `if - else` szerkezettel megvalósított változata. Egy másik példa*:

```
printf("You have %d item%s\n", n, n==1 ? "" : "s");
```

2.10. gyakorlat. Írjuk át a nagybetűket kisbetűkké alakító `lower` függvényt úgy, hogy az `if - else` szerkezetet feltételes kifejezéssel helyettesítjük!

2.12. A precedencia és a kifejezés kiértékelési sorrendje

A 2.1. táblázatban összefoglaltuk az összes operátor (beleértve az eddig nem tárgyaltakat is) precedenciáját és asszociativitását (kiértékelési irányát). Az azonos sorban szereplő operátorok precedenciája azonos, a sorok csökkenő precedencia szerint követik egymást, és így pl. a `*`, `/` és `%` precedenciája azonos és nagyobb a két-operandusú + és - precedenciájánál. A táblázatban a `( )` operátor a függvényhívást jelöli, a `->` és `.`

* Az eddigiektől eltérően ezt a példát az angol és a magyar nyelvtan eltérő volta miatt nem tudtuk magyarra átírni. A példa lényege, hogy a programrész az *n* szám 1 vagy 1-nél nagyobb értékétől függően a mondatot „You have 1 item." vagy „You have 5 items." alakban írja ki (ha pl. *n*=5), azaz az „item" szó egyes vagy többes számát használja az angol nyelvtani szabályoknak megfelelően. (A mondat magyarul: „1 árucikked van", ill. „5 árucikked van".) (A fordító)

2.1. táblázat. Operátorok precedenciája és asszociativitása

Operátor	Asszociativitás
`() [] ->.`	balról jobbra
`! ~ ++ -- + - * & (típus) sizeof`	jobbról balra
`* / %`	balról jobbra
`+ -`	balról jobbra
`<< >>`	balról jobbra
`< <= > >=`	balról jobbra
`== !=`	balról jobbra
`&`	balról jobbra
`^`	balról jobbra
`|`	balról jobbra
`&&`	balról jobbra
`||`	balról jobbra
`?:`	jobbról balra
`= += -= *= /= %= &= ^= |= <<= >>=,`	jobbról balra
	balról jobbra

Megjegyzés: az unáris (egyoperandusú) +, – és * operátorok nagyobb precedenciájúak, mint a kétoperandusú operátorok.

operátorok pedig struktúrák elemeihez való hozzáféréshez használhatók. Ezekkel a 6. fejezetben foglalkozunk, csakúgy mint az objektum méretét megadó `sizeof` operátorral. A mutatókkal kapcsolatos indirekciós `(*)` operátort és az objektum címét megadó `&` operátort az 5. fejezetben, a `,` (vessző) operátort pedig a 3. fejezetben tárgyaljuk.

Megjegyezzük, hogy a bitenkénti `&`, `^` és `|` operátorok precedenciája ilyen sorrendben csökken és eleve kisebb az `==` és `!=` precedenciájánál. Emiatt az olyan bitvizsgáló kifejezéseket, mint

```
if ((x & MASZK) == 0) ...
```

a helyes eredmény érdekében gondosan zárójelezni kell.

A C, több más nyelvhez hasonlóan, nem határozza meg az operátorok operandusainak kiértékelési sorrendjét. (Kivéve az `&&`, `||`, `?:` és `,` operátorokat.) Így pl. felmerül a kérdés, hogy az

```
x = f( ) + g( );
```

alakú kifejezésben az `f` kiértékelése a `g` előtt történik-e vagy sem. Ez azért érdekes, mert `f` vagy `g` megváltoztathatja a másik függvényben használt változókat, és emiatt `x` értéke függhet a kiértékelés sorrendjétől. Ha valami miatt fontos a kiértékelés sorrendje, akkor a közbenső eredményeket egy segédváltozóban kell eltárolni és később felhasználni.

Hasonlóan a függvényargumentumok kiértékelési sorrendje sincs meghatározva, ezért a

```
printf("%d %d\n", ++n, power(2, n)); /* hibás */
```

utasítás különböző fordítóprogramokkal más-más eredményt adhat, aszerint, hogy az `n` inkrementálása a `power` függvény hívása előtt vagy után történik-e. A biztos megoldás természetesen az, ha az utasítást

```
++n;
printf("%d %d\n", n, power(2, n));
```

alakban írjuk.

A függvényhívások, egymásba ágyazott értékadó utasítások és inkrementáló, ill. dekrementáló utasítások mellékhatásokat okozhatnak, azaz egy kifejezés kiértékelése közben néhány változó értéke (nem szándékosan) megváltozhat. Bármely kifejezés kiértékelése vezethet mellékhatásokhoz, és sok függhet attól, hogy a kifejezésben aktuálisan szereplő változókat milyen sorrendben dolgozza fel a gép. Egy szerencsétlen, de sajnos gyakori esetet példáz a következő utasítás:

```
a[i] = i++;
```

A kérdés az, hogy az index a régi vagy az új `i` érték-e? A fordítóprogramok a kérdést különböző módon kezelik, és az eredmény a felhasznált fordítóprogramtól függ. A szabvány a legtöbb ilyen kérdést szándékosan nyitva hagyja. A kifejezések kiértékelése során a mellékhatások kialakulását végezetül is mindig a fordítóprogram dönti el, mivel az optimális kiértékelési sorrend nagymértékben függ a számítógép felépítésétől.

A kiértékelési sorrendtől függő programok írása minden nyelv esetén helytelen, ezért mindenképpen kerüljük és igyekezzünk kiküszöbölni a mellékhatásokat is. A mellékhatások elkerüléséhez jó, ha tudjuk, hogy adott helyzetben *mit* csinál a számítógép (vagy a fordítóprogram), viszont a tudatlanság sok esetben védelmet is jelenthet.

Vezérlési szerkezetek

Egy nyelv vezérlésátadó utasításai az egyes műveletek végrehajtási sorrendjét határozzák meg. A korábbi példákban már találkoztunk a legfontosabb vezérlési szerkezetekkel. Ebben a fejezetben teljessé tesszük a képet és a korábbiaknál pontosabban írjuk le az egyes szerkezetek tulajdonságait.

3.1. Utasítások és blokkok

Egy olyan kifejezés, mint `x = 0, i++` vagy `printf(...)` *utasítássá* válik, ha egy pontosvesszőt írunk utána. Pl.:

```
x = 0;
i++;
printf(...);
```

A C nyelvben a pontosvessző az utasításlezáró jel (terminátor), szemben a Pascal nyelvvel, ahol elválasztó szerepe van.

A { } kapcsos zárójelekkel deklarációk és utasítások csoportját fogjuk össze egyetlen *összetett utasításba* vagy *blokkba,* ami szintaktikailag egyenértékű egyetlen utasítással. A kapcsos zárójelek használatára jó példa a függvények utasításait összefogó zárójelpár, vagy az `if, else, while, for,` ill. hasonló utasítások utáni utasításokat összefogó zárójelpár. A változók *bármelyik* blokk belsejében deklarálhatók, erről bővebben a 4. fejezetben írunk. A blokk végét jelző jobb kapcsos zárójel után soha nincs pontosvessző.

3.2. Az if–else utasítás

Az `if-else` utasítást döntés kifejezésére használjuk. Formálisan az utasítás szintaxisa a következő:

```
if (kifejezés)
    1. utasítás
else
    2. utasítás
```

ahol az `else` rész opcionális. Az utasítás először kiértékeli a kifejezést, és ha ennek értéke igaz (azaz a kifejezés értéke nem nulla), akkor az 1. utasítást hajtja végre. Ha a kifejezés értéke hamis (azaz nulla) és van `else` rész, akkor a 2. utasítás hajtódik végre.

Mivel az `if` egyszerűen csak a kifejezés számértékét vizsgálja, ezért lehetőség van a program rövidítésére. A legnyilvánvalóbb ilyen lehetőség, ha az

```
if (kifejezés != 0)
```

helyett az

```
if (kifejezés)
```

utasítást írjuk. Egyes esetekben ez a forma természetes és nyilvánvaló, máskor viszont elég áttekinthetetlenné teszi a programot.

Mivel az `if-else` szerkezet `else` ága opcionális egymásba ágyazott `if-else` szerkezeteknél, hiányzó `else` ágak esetén nem világos, hogy a meglévő `else` ág melyik `if` utasításhoz tartozik. Például az

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

programrészletben az `else` a belső `if` utasításhoz tartozik, amit a programrész tagolása is mutat. Általános szabályként megfogalmazhatjuk, hogy az `else` mindig a hozzá legközelebb eső, `else` ág nélküli `if` utasításhoz tartozik. Ha nem így szeretnénk, akkor a kívánt összerendelés kapcsos zárójelekkel érhető el, mint pl. az

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

szerkezetben. A nem egyértelmű helyzet különösen zavaró az olyan szerkezetekben, mint az

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else /* ez így hibás */
    printf("hiba: n értéke negatív\n");
```

A tagolás egyértelműen mutatja a szándékot, de a fordítóprogram ezt nem veszi figyelembe, és minden figyelmeztető jelzés nélkül a belső `if` utasításhoz kapcsolja az `else` ágat. Az ilyen hibák nagyon nehezen deríthetők fel, és legegyszerűbben úgy kerülhetjük el azokat, ha a beágyazott `if` utasítást kapcsos zárójelekkel határoljuk.

Vegyük észre, hogy a `z = a` értékadás után az

```
if (a > b)
    z = a;
else
    z = b;
```

szerkezetben pontosvessző van. Ennek az az oka, hogy az `if` utasítást egy újabb *utasítás* követi (a `z = a`), amit pontosvesszővel kell zárni.

3.3. Az else–if utasítás

Az

```
if (kifejezés)
    utasítás
else if (kifejezés)
    utasítás
else if (kifejezés)
    utasítás
else if (kifejezés)
    utasítás
.
.
.
else
    utasítás
```

szerkezet olyan gyakran fordul elő, hogy mindenképpen megérdemli a részletesebb elemzést. Ez a szerkezet adja a többszörös döntések (elágazások) programozásának egyik legáltalánosabb lehetőségét. A szerkezet úgy működik, hogy a gép sorra kiértékeli a *kifejezéseket* és ha bármelyik ezek közül igaz, akkor végrehajtja a megfelelő *utasítást*, majd befejezi az egész vizsgáló láncot. Itt is, mint bárhol hasonló esetben, az utasítás helyén kapcsos zárójelek között elhelyezett blokk is állhat.

A szerkezet utolsó (`if` nélküli) `else` ága alapértelmezés szerint a „fentiek közül egyik sem" esetet kezeli. Néha ilyenkor semmit sem kell csinálni, ezért a szerkezetet záró

```
else
    utasítás
```

ág hiányozhat, vagy valamilyen lehetetlen eset érzékelésével hibaellenőrzésre használható.

A következő példában egy háromirányú elágazás (döntés) látható. A feladat, hogy írjunk egy bináris keresést végző függvényt, amely egy rendezett `v` tömbben megkeresi az `x` értéket és megadja annak helyét (indexét). A függvény `x` előfordulási helyével (ami 0 és `n-1` közé eshet) tér vissza, ha `x` megtalálható `v` elemei között és `-1` értékkel, ha nem.

A bináris keresési algoritmus az `x` bemeneti értéket először összehasonlítja a rendezett `v` tömb középső elemével. Ha `x` kisebb a középső elemnél, akkor a keresés a táblázat alsó felében, ha nem, akkor a felső felében folytatódik. Mindkét esetben a következő lépés, hogy `x`-et összehasonlítjuk a táblázat megfelelő felében (alsó vagy felső) lévő középső elemmel. A két egyenlő részre osztás addig folytatódik, amíg a keresett elemet meg nem találtuk, vagy a tartomány mérete nulla nem lesz (üres tartomány). A `binsearch` kereső függvénynek három argumentuma lesz, a keresett `x`, a rendezett `v` tömb és a tömb elemeinek `n` száma. A program:

```
/* binsearch: megkeresi x értékét a növekvő irányba
rendezett v[0]...v[n-1] tömbben */
int binsearch(int x, int v[ ], int n)
```

```
{
    int also, felso, kozep;

    also = 0;
    felso = n - 1;
    while (also <= felso) {
        kozep = (also + felso) / 2;
        if (x < v[kozep])
            felso = kozep - 1;
        else if (x > v[kozep])
            also = kozep + 1;
        else /* megtalálta */
            return kozep;
    }
    return -1; /* nem találta meg */
}
```

A program minden lépésében az alapvető döntés az, hogy x kisebb, nagyobb vagy egyenlő a v[kozep] középső eleménél, és ezt a döntési sort egyszerűen valósíthatjuk meg az else-if szerkezettel.

3.1. gyakorlat. A bináris kereső program a ciklus belsejében két vizsgálatot végez, de egy is elegendő lenne (a futási idő csökkentése érdekében érdemes minden lehetséges műveletet a ciklusmagon kívül elvégezni). Írja meg a program olyan változatát, amelyben a cikluson belül csak egy vizsgálat van és hasonlítsa össze a kétféle változat futási idejét!

3.4. A switch utasítás

A switch utasítás is a többirányú programelágaztatás egyik eszköze. Az utasítás úgy működik, hogy összehasonlítja egy kifejezés értékét több egész értékű állandó kifejezés értékével, és az ennek megfelelő utasítást hajtja végre. A switch utasítás általános felépítése:

```
switch (kifejezés) {
    case állandó kifejezés: utasítások
    case állandó kifejezés: utasítások
    .
    .
    .
    default: utasítások
}
```

Mindegyik case ágban egy egész állandó vagy állandó értékű kifejezés található, és ha ennek értéke megegyezik a switch utáni kifejezés értékével, akkor végrehajtódik a case ágban elhelyezett egy vagy több utasítás. Az utolsó, default ág akkor hajtódik végre, ha egyetlen case ághoz tartozó feltétel sem teljesült. A default ág opcionális, ha elhagyjuk és a case ágak egyike sem teljesül, akkor semmi sem történik. A case ágak és a default ág tetszőleges sorrendben követhetik egymást.

Az 1. fejezetben leírtunk egy olyan programot, amely megszámolta az egyes számjegyek, üres helyek és más karakterek előfordulását. A programban if-else if-else szerkezetet használtunk, most elkészítjük a switch utasítással felépített változatát.

```
#include <stdio.h>

main( ) /* számok, üres helyek és mások számolása */
{
    int c, i, nures, nmas, nszam[10];

    nures = nmas = 0;
    for (i = 0; i < 10; i++)
        nszam[i] = 0;
    while ((c = getchar( )) != EOF)

        switch (c) {
        case '0': case '1': case '2': case '3':
        case '4': case '5': case '6': case '7':
        case '8': case '9':

            nszam[c-'0']++;
            break;
        case ' ':
        case '\':
        case '\t':
            nures++;
            break;
        default:
            nmas++;
            break;
        }
    }
    printf("számok =");
    for (i = 0; i < 10; i++)
        printf(" %d", nszam[i]);
    printf(", üres hely = %d, más = %d\n", nures, nmas);
    return 0;
}
```

A `break` utasítás hatására a vezérlés azonnal abbahagyja a további vizsgálatokat és kilép a `switch` utasításból. Az egyes `case` esetek címkeként viselkednek, és miután valamelyik `case` ág utasításait a program végrehajtotta, a vezérlés azonnal a következő `case` ágra kerül, hacsak explicit módon nem gondoskodunk a kilépésről. A `switch` utasításból való kilépés legáltalánosabb módja a `break` és `return` utasítással való kilépés. A `break` utasítás a `while`, `for` vagy `do` utasításokkal szervezett ciklusokból való kilépésre is használható, amint erről a későbbiekben még szó lesz.

Az egyes `case` ágakon való folyamatos végighaladás nem egyértelműen előnyös. Pozitív oldala, hogy több esethez azonos tevékenység rendelhető (mint a példában, ahol az összes számjegyhez azonos tevékenységet rendeltünk). De ebből következik, hogy minden `case` ágat `break` utasítással kell lezárni, nehogy a vezérlés a következő `case` ágra kerüljön. Az egyik `case` ágról a másikra való lépkedés nem túl ésszerű, mert a program módosítása esetén a vezérlés széteshet. Azokat az eseteket leszámítva, amikor több `case` ághoz közös tevékenység tartozik, érdemes kerülni az egyes `case` ágak közötti átmenetet.

A program áttekinthetősége érdekében akkor is helyezzünk el `break` utasítást a programban, ha az utolsó ágban vagyunk (mint a példában, ahol a `default` ág is tartalmaz

break utasítást), annak ellenére, hogy ez logikailag szükségtelen. Ha valamikor később a vizsgálati sort újabb case ágakkal egészítjük ki, ez a defenzív programozási stílus előnyös lesz.

3.2. gyakorlat. Írjunk escape(s, t) néven függvényt, amely a t karaktersorozatot az s karaktersorozat végéhez másolja és a másolás során a láthatatlan karaktereket (pl. új sor, tabulátor) látható escape sorozatokká (\n, \t) alakítja! A programot a switch utasítással írjuk meg! Készítsük el a függvény inverzét is, amely az escape sorozatokat a tényleges karakterekké alakítja!

3.5. Ciklusszervezés while és for utasítással

Korábban már találkoztunk a while és for utasításokkal szervezett ciklusokkal. A

```
while (kifejezés)
    utasítás
```

szerkezetben a program először kiértékeli a *kifejezést*. Ha annak értéke nem nulla (igaz), akkor az utasítást végrehajtja, majd a kifejezés újra kiértékelődik. Ez a ciklus mindaddig folytatódik, amíg a *kifejezés* nullává (hamissá) nem válik, és ilyen esetben a program végrehajtása az *utasítás utáni* helyen folytatódik.

A for utasítás általános szerkezete:

```
for (1. kifejezés; 2. kifejezés; 3. kifejezés)
    utasítás
```

ami teljesen egyenértékű a while utasítással megvalósított

```
1. kifejezés;
while (2. kifejezés) {
    utasítás
    3. kifejezés;
}
```

szerkezettel, kivéve a continue utasítás viselkedését, amivel a 3.7. pontban foglalkozunk.

Szintaktikailag a for utasítás mindhárom komponense kifejezés. Leggyakrabban az *1.* és *3. kifejezés* értékadás vagy függvényhívás, és a *2. kifejezés* egy relációs kifejezés. A három komponens bármelyike hiányozhat, de az őket lezáró pontosvessző kiírása ekkor is kötelező. Ha az *1.* vagy *3. kifejezés* hiányzik, akkor azokat egyszerűen elhagyjuk a for utasítást követő zárójelből. Ha a *2.* (vizsgáló) *kifejezés* is hiányzik, akkor azt a gép úgy tekinti, hogy az állandóan igaz, és ezért a

```
for (;;) {
    ...
}
```

szerkezet egy végtelen ciklus, amiből feltehetőleg más módon (pl. break vagy return utasítással) kell kilépni.

Teljesen a programozóra van bízva, hogy mikor használ while és mikor for utasítást a ciklusszervezéshez. Például a

```
while ((c = getchar( )) == ' ' || c == '\n' ||
       c == '\t')
   ; /* ugorjon az üres karaktereknél */
```

esetén nincs kezdeti értékadás vagy újbóli értékadás, ezért a `while` használata elég kézenfekvő.

A `for` utasítás egyszerű inicializálás és újrainicializálás esetén előnyös, mivel a ciklust vezérlő utasítások együtt, jól látható formában, a ciklusmag tetején helyezkednek el. Ez jól látszik a

```
for (i = 0; i < n; i++)
   ...
```

szerkezetben, ami egyébként pl. egy tömb első `n` elemét feldolgozó programrész C nyelvű megfogalmazása. A programrész hasonló a FORTRAN DO utasításával vagy a Pascal `for` utasításával szervezett ciklushoz, annyi eltéréssel, hogy a C nyelvű `for` ciklusban a ciklusváltozó és a ciklus határa a ciklus belsejében változtatható, valamint hogy a ciklusváltozó a ciklusból való kilépés esetén is megtartja az értékét. Mivel a `for` ciklus komponensei tetszőleges kifejezések lehetnek, a `for` ciklus nem korlátozódik aritmetikai léptetésekre. Stiláris szempontból mégis helytelen, ha a `for` utasítás inicializáló és inkrementáló részébe a `for`-tól idegen számításokat helyezünk el. Célszerű a `for` utasítást kizárólag a ciklus vezérlésére fenntartani.

Nagyobb példaként bemutatjuk a számjegyekből álló karaktersorozatot számmá alakító `atoi` függvény egy másik változatát. Ez kissé általánosabb a 2. fejezetben bemutatott változatnál: képes a karaktersorozat bevezető üres helyeinek és a szám esetleg kiírt `+` vagy `-` előjelének kezelésére is. (A 4. fejezetben ismertetjük az `atof` függvényt, ami ugyanezt a feladatot lebegőpontos számokkal valósítja meg.)

A program szerkezete a bemeneti adatok struktúráját tükrözi:

ugord át az üres helyeket, ha vannak
olvasd be az előjelet, ha van
olvasd be az egészrészt és konvertáld

Minden programrész elvégzi a maga feladatát és a dolgokat „tiszta" állapotban adja át a következő programrésznek. Az egész folyamat akkor ér véget, ha beolvasódik az első olyan karakter, ami nem lehet része egy egész számnak.

```
#include <ctype.h>

/* atoi: az s karaktersorozat számmá alakítása */

int atoi(char s[ ])
{
   int i, n, sign;

   for (i = 0; isspace(s[i]); i++)
      ; /* átugorja az üres helyeket */
   sign = (s[i] == '-') ? -1 : 1;
   if (s[i] == '+' || s[i] == '-')
      i++; /* átugorja az előjelet */
   for (n = 0; isdigit(s[i]); i++)
```

```
        n = 10 * n + (s[i]-'0');
    return sign*n;
}
```

A standard könyvtár a sokkal részletesebben kimunkált `strtol` függvényt tartalmazza, ami karaktersorozatok `long` típusú egésszé alakítására használható. Ennek leírását a B. Függelék 5. pontjában találjuk meg.

A ciklus vezérlésének egy helyen tartása főleg akkor előnyös, ha több, egymásba ágyazott ciklus van. Ezt jól példázza a következő program, amely az egész számokból álló tömb elemeit rendezi a Shell-algoritmussal. A rendezési algoritmus (amelyet D. L. Shell dolgozott ki 1959-ben) alapgondolata, hogy kezdetben az egymástól távoli elemek kerülnek összehasonlításra, szemben az egyszerűbb, cserélgetős rendezési algoritmusokkal, ahol a szomszédos elemeket hasonlítják össze. Ezáltal a kezdetben meglévő nagyfokú rendezetlenség gyorsan csökken és a későbbi lépésekben kevesebb munkát kell végezni. A programban az éppen összehasonlított elemek közti távolság fokozatosan egyre csökken, és végül a rendezés az egyszerű, szomszédos elemeket cserélgető rendezésbe megy át.

```
/* shellsort: a v[0]...v[n-1] tömb rendezése
növekvő sorrendbe */
void shellsort(int v[ ], int n)
{
    int tavolsag, i, j, atm;

    for (tavolsag = n/2; tavolsag > 0; tavolsag /= 2)
        for (i = tavolsag; i < n; i++)
            for (j = i-tavolsag; j >= 0 && v[j] >
                v[j + tavolsag];
                j -= tavolsag) {
                    atm = v[j];
                    v[j] = v[j+tavolsag];
                    v[j+tavolsag] = atm;
            }
}
```

A programban három egymásba ágyazott ciklus van. A legkülső az összehasonlítandó elemek közötti távolságot szabályozza úgy, hogy azt `n/2` értékről indítva minden lépésben a felére csökkenti, egészen addig, amíg nulla nem lesz. A középső ciklus folyamatosan végigmegy az elemeken. A legbelső ciklus összehasonlítja az egymástól `tavolsag` értékre lévő elemeket és ha nincsenek megfelelő sorrendben, akkor megcseréli azokat. Mivel a `tavolsag` az utolsó lépésben 1-re csökken, ezért a tömb végül is helyes sorrendbe rendeződik. Vegyük észre, hogy a külső ciklus `for` utasítása ugyanolyan alakú, mint a többi, bár ez a `for` utasítás nem végez aritmetikai léptetést.

A C nyelv egyik eddig még nem említett operátora a , (vessző), amelyet legtöbbször a `for` utasításban használunk. A vesszővel elválasztott kifejezéspárok balról jobbra értékelődnek ki, és az eredmény típusa, ill. értéke a jobb oldalon álló operandus típusával, ill. értékével egyezik meg. Így egy `for` utasítás lehetőséget ad az egyes részekben több kifejezés elhelyezésére, pl. két index szerint párhuzamosan végzett feldolgozás érdekében. Ezt a megoldást példázza a `reverse(s)` függvény, amelynek feladata, hogy az `s` karaktersorozatot saját helyén megfordítsa.

```
#include <string.h>
/* reverse: az s karaktersorozat megfordítása helyben */
void reverse(char s[ ])
{
    int c, i, j;

    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

A függvények argumentumait, a deklarációban lévő változókat stb. elválasztó vessző nem vesszőoperátor, nem garantált a balról jobbra irányú feldolgozásuk.

A vesszőoperátort viszonylag ritkán használják, és a legtöbb alkalmazás szoros kapcsolatban van egymással. Ilyen pl. a `reverse` függvényben a `for` ciklus vagy a makróban a többlépéses számítások egyetlen kifejezéssel történő megadása. A vessző operátor jól használható a `reverse` függvényben az egyes tömbelemek cseréjénél is. A

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

szerkezetben a cserélés műveletei egyetlen utasításba foghatók össze.

3.3. gyakorlat. Írjunk `expand(s1, s2)` néven függvényt, amely az `s1` karaktersorozatban lévő rövidítéseket `s2` karaktersorozatban feloldja (pl. az `a-z` helyett kiírja az `abc...xyz` teljes listát)! A program tegye lehetővé a betűk és számjegyek kezelését, és gondoljunk olyan rövidítések feloldására is, mint `a-b-c`, `a-z0-9` vagy `-a-z` is! Célszerű a kezdő vagy záró - jelet literálisként kezelni.

3.6. Ciklusszervezés do–while utasítással

Amint azt már az 1. fejezetben elmondtuk, a `while` és a `for` utasítással szervezett ciklusok közös tulajdonsága, hogy a ciklus leállításának feltételét a ciklus tetején (a ciklusmagba belépés előtt) vizsgálják. Ezzel ellentétesen működik a C nyelv harmadik ciklusszervező utasítása, a `do-while` utasítás. A `do-while` utasítás a ciklus leállításának feltételét a ciklusmag végrehajtása után ellenőrzi, így a ciklusmag egyszer garantáltan végrehajtódik. Az utasítás általános formája:

```
do
    utasítás
while (kifejezés);
```

A gép először végrehajtja az *utasítást* és csak utána értékeli ki a *kifejezést.* Ha a kifejezés értéke igaz, az utasítás újból végrehajtódik. Ez így megy mindaddig, amíg a kifejezés értéke hamis nem lesz, ekkor a ciklus lezárul és a végrehajtás az utána következő utasítással folytatódik. Az ellenőrzés módjától eltekintve a `do-while` utasítás egyenértékű a Pascal `repeat-until` utasításával.

A tapasztalatok azt mutatják, hogy a `do-while` utasítást sokkal ritkábban használ-

ják, mint a `while` vagy `for` utasítást, bár hasznos tulajdonságai miatt időről időre célszerű elővenni, mint pl. a következőkben bemutatott `itoa` függvényben, amely egy számot karaktersorozattá alakít (az `atoi` függvény inverze). A feladat kicsit bonyolultabb, mint elsőre látszik, mivel a számjegyeket generáló egyszerű megoldások rossz sorrendet eredményeznek. Ezért úgy döntöttünk, hogy a karakterláncot fordított sorrendben generáljuk, majd a végén megfordítjuk.

```
/* itoa: az n számot s karaktersorozattá alakítja */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0) /* elteszi az előjelet */
        n = -n; /* n-et pozitívvá teszi */
    i = 0;
    do { /* generálja a számjegyeket, de fordított sor-
            rendben */
        s[i++] = n % 10 + '0'; /* a következő számjegy */
    } while ((n /= 10) > 0); /* törli azt */
    if (sign < 0)

        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

A példában a `do-while` használata szükségszerű, vagy legalábbis kényelmes, mivel legalább egy karaktert akkor is el kell helyeznünk az `s` karaktersorozatban, ha `n` értéke nulla. A `do-while` ciklus magját kapcsos zárójellel kiemeltük (bár szükségtelen), mivel így a `while` rész nem téveszthető össze egy `while` utasítással szervezett *ciklus kezde-tével.*

3.4. gyakorlat. Az `itoa` függvény itt ismertetett változata kettes komplemens kódú számábrázolás esetén nem kezeli a legnagyobb negatív számot; azaz az n = –2↑(szóhossz–1) értéket. Magyarázzuk meg, hogy miért! Módosítsuk úgy a programot, hogy ezt az értéket is helyesen írja ki, a használt számítógéptől függetlenül.

3.5. gyakorlat. Írjunk `itob(n, s, b)` néven függvényt, amely az `n` egész számot `b` alapú számrendszerben karaktersorozattá alakítja és az `s` karaktersorozatba helyezi! Speciális esetként írjuk meg az `itob(n, s, 16)` függvényt is, amely az `n` értékét hexadecimális formában írja az `s` karaktersorozatba.

3.6. gyakorlat. Írjuk meg az `itoa` függvénynek azt a változatát, amelynek kettő helyett három argumentuma van! Ez a harmadik argumentum legyen a minimális mezőszélesség, és az átalakított számot szükség esetén balról üres helyekkel töltse fel, hogy elegendően széles legyen!

3.7. A break és continue utasítások

Néha kényelmes lehet, ha egy ciklusból az elején vagy végén elhelyezett ellenőrzés kikerülésével is ki tudunk lépni. A `break` utasítás lehetővé teszi a `for`, `while` vagy `do` utasításokkal szervezett ciklusok idő előtti elhagyását, valamint a `switch` utasításból

való kilépést. A break hatására a legbelső ciklus vagy a teljes switch utasítás fejeződik be.

A működés bemutatására írjuk meg a trim függvényt, amely egy karaktersorozat végéről eltávolítja a szóközöket, tabulátorokat és újsor-karaktereket. A program a break utasítást használja a ciklusból való idő előtti kilépésre, ha megtalálja a karaktersorozat legjobboldalibb (utolsó) nem szóköz-, tabulátor- vagy újsor-karakterét.

```
/* trim: eltávolítja a záró szóköz-, tabulátor- vagy
újsor-karaktereket */
int trim(char s[ ])
{
    int n;
    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' &&
            s[n] != '\n')
                break;
    s[n + 1] = '\0';
    return n;
}
```

Az strlen függvény visszatéréskor a karaktersorozat hosszát adja meg. A for ciklus a karaktersorozat végén kezdi a feldolgozást és addig vizsgálja a karaktereket, amíg megtalálja az első szóköztől, tabulátortól vagy új sortól különböző karaktert. A ciklus akkor áll le, ha az első ilyen karaktert megtaláltuk vagy n értéke negatívvá válik (azaz, amikor a teljes karaktersorozatot végignéztük). Az olvasóra bízzuk, hogy igazolja, a program akkor is helyesen működik, ha a karaktersorozat üres vagy csak üres helyet adó karaktereket tartalmaz.

A continue utasítás a break utasításhoz kapcsolódik, de annál ritkábban használjuk. A ciklusmagban található continue utasítás hatására azonnal (a ciklusmagból még hátralévő utasításokat figyelmen kívül hagyva) megkezdődik a következő iterációs lépés. A while és do utasítások esetén ez azt jelenti, hogy azonnal végbemegy a feltételvizsgálat, for esetén pedig a ciklusváltozó újrainicializálódik. A continue utasítás csak ciklusokban alkalmazható, a switch utasításban nem. Ha a switch utasításban ciklus volt, akkor a continue ezt a ciklust lépteti tovább.

A continue működését illusztráló egyszerű programrész az a tömb nem negatív elemeit dolgozza fel, a negatív elemeket átugorja.

```
for (i = 0; i < n; i++) {
    if (a[i] < 0) /* a negatív elemek átugrása */
        continue;
    ... /* a pozitív elemek feldolgozása */
}
```

A continue utasítást gyakran használjuk olyan esetekben, amikor a ciklus további része nagyon bonyolult, és ezért a vizsgálati feltétel megfordítása, ill. egy újabb programszint beágyazása a programot túlzottan mélyen tagolná.

3.8. A goto utasítás és a címkék

A C nyelvben is használható a gyakran szidott `goto` utasítás, amellyel megadott címkékre ugorhatunk. Alapvetően a `goto` utasításra nincs szükség és a gyakorlatban majdnem mindig egyszerűen írhatunk olyan programot, amelyben nincs `goto`. A könyvben közölt mintaprogramok a továbbiakban sem tartalmaznak `goto` utasítást.

Mindezek ellenére most bemutatunk néhány olyan esetet, amelyben a `goto` utasítás működése megfigyelhető. A `goto` használatának egyik legelterjedtebb esete, amikor több szinten egymásba ágyazott szerkezet belsejében kívánjuk abbahagyni a feldolgozást és egyszerre több, egymásba ágyazott ciklusból szeretnénk kilépni. Ilyenkor a `break` utasítás nem használható, mivel az csak a legbelső ciklusból lép ki.

Például a

```
for ( ... )
    for ( ... ) {

        ...
        if (zavar)
            goto hiba;
    }
...
hiba:
    a hiba kezelése
```

szerkezetben előnyös a hibakezelő eljárást egyszer megírni és a különböző hibaeseteknél a vezérlést a közös hibakezelő eljárásnak átadni, bárhol is tartott a feldolgozás.

A címke ugyanolyan szabályok szerint alakítható ki, mint a változók neve és mindig kettőspont zárja. A címke bármelyik utasítás előtt állhat és a `goto` utasítással bármelyik, a `goto`-val azonos függvényben lévő utasítás elérhető. A címke hatásköre arra a teljes függvényre kiterjed, amiben használják.

Második példaként tekintsük azt a feladatot, amikor meg szeretnénk határozni, hogy az `a` és `b` tömbnek vannak-e közös elemei. Egy lehetséges megoldás:

```
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            if (a[i] == b[j])
                goto talalt;
    /* nem talált közös elemet */
    ...
talalt:
    /* egy közös elem van, a[i] == b[j] */
    ...
```

Mint említettük, minden `goto`-t tartalmazó program megírható `goto` nélkül is, bár ez néha bonyolult a sok ismétlődő vizsgálat és segédváltozó miatt. Az előző példa `goto` nélküli változata:

```
talalt = 0;
for (i = 0; i < n && !talalt; i++)
    for (j = 0; j < m && !talalt; j++)
```

```
            if (a[i] == b[j])
               talalt = 1;
    if (talalt)
       /* egy közös elem van, a[i-1] == b[j-1] */
       ...
else
       /* nem talált közös elemet */
       ...
```

Néhány itt bemutatott kivételtől eltekintve a `goto` utasítást tartalmazó programok általában nehezebben érthetők és kezelhetők, mint a `goto` nélküli programok. Bár ez nem törvény, de jó ha betartjuk: a `goto` utasítást a lehető legritkábban használjuk.

4. FEJEZET:

Függvények és a program szerkezete

A függvényekkel a nagyobb számítási feladatok kisebb egységekre oszthatók, így a programozó felhasználhatja a már meglévő egységeket és nem kell minden alkalommal elölről kezdeni a munkát. A függvények a működésük részleteit gyakran elrejtik a program többi része elől, de jól megírt függvények esetén nincs is szükség ezekre a részletekre.

A C nyelvet úgy tervezték meg, hogy a függvények hatékonyak és jól használhatók legyenek. A C nyelvű programok sokkal inkább több, kisebb függvényből állnak, mint egy nagyobból. A program egy vagy több forrásállományban helyezkedhet el. A forrásállományok egymástól függetlenül fordíthatók és a már korábban lefordított könyvtári függvényekkel együtt tölthetők be. Ennek menete az alkalmazott operációs rendszertől függ, ezért pillanatnyilag nem foglalkozunk vele.

Az ANSI szabvány a függvények deklarációja és definíciója terén változtatta meg leginkább a C nyelvet. Mint az 1. fejezetben már említettük, lehetővé vált az argumentumok típusának deklarálása a függvény deklarálásával egyidejűleg. A függvény definiálásának szintaxisa szintén megváltozott, ami lehetővé teszi a fordítóprogramnak, hogy a korábbi változatokhoz képest sokkal több hibát felderítsen. Az új szabvány további előnye, hogy helyesen deklarált argumentumok esetén létrejön a kényszerített automatikus típuskonverzió.

A szabvány tisztázza a nevek hatáskörének kérdését is, különösen azzal, hogy megköveteli az egyes külső objektumok egyszeri definícióját. Az inicializálás teljesen általánosan működik, és az automatikus tárolási osztályú tömbök, ill. struktúrák egyszerűen inicializálhatók.

A bevezetőben fontos megemlíteni, hogy a C előfeldolgozó rendszert is továbbfejlesztették. A feltételes fordítási direktívák korábbinál sokkal teljesebb készlete új lehetőségekkel bővült, ami elsősorban a makrokifejtés hatékonyabb vezérlésében jelentkezik, valamint abban, hogy a makro argumentumából aposztófokkal határolt karaktersorozat generálható.

4.1. A függvényekkel kapcsolatos alapfogalmak

Az alapfogalmak bemutatásához írjunk egy programot, amely a bemenetére adott szöveg minden olyan sorát kiírja, amiben megtalálható egy adott minta (karaktersorozat). A program a UNIX `grep` segédprogramjának egy speciális változata. Például a következő szövegben keressük a „dal" mintát az egyes sorokban.*

Óh lakodalmi kar-dal,
éneklő diadal,
ki is verseng e dallal,
oly édes, fiatal,
hiába itt a harc, meddő a viadal.

* A feladatot (mint általában eddig is) magyarítottuk. A példában szereplő versszak Percy Bysshe Shelley: Egy mezei pacsirtához című verséből van, Kosztolányi Dezső fordításában. (A fordító)

A program eredményül a következő szöveget hozza létre:

Óh lakodalmi kar-dal,
éneklő diadal,
ki is verseng e dallal,
hiába itt a harc, meddő a viadal.

A feladat három alapvető részre osztható, és így a program szerkezete:

```
while  (van további sor)
    if  (a sor tartalmazza a mintát)
        nyomtassuk ki
```

Bár a teljes programot egyetlen egységként (egyetlen `main` függvényként) is megírhatjuk, mégis célszerű a fenti programszerkezet előnyeit kihasználni és az egyes részeket önálló függvényként megírni. A három kis résszel könnyebben megbirkózunk, mint az egyetlen naggyal, mivel a lényegtelen részletkérdések „belevesznek" az egyes függvényekbe és a nem kívánt kölcsönhatások esélye minimális lesz. Ezenkívül bármely programrészt (függvényt) más programokban is felhasználhatjuk.

A „van még további sor" feladatrészt az 1. fejezetben már megírt `getline` függvénnyel, a „nyomtassuk ki" feladatrészt pedig a `printf` függvénnyel valósítjuk meg. Ez azt jelenti, hogy csak azt a programrészt kell megírnunk, amely eldönti, hogy a keresett minta megtalálható-e a vizsgált sorban.

Ezt a programrészt az `strindex(s, t)` függvénnyel oldjuk meg. A `strindex` függvény első argumentuma (`s`) a vizsgált sort tartalmazó karaktertömb, a második argumentuma (`t`) a keresett mintát tartalmazó karaktertömb, és visszatérési értéke a `t` tömb kezdetének `s` tömbbeli indexe, vagy –1, ha `s` nem tartalmazza a keresett mintát. Mivel egy C-beli tömb kezdetének indexe nulla, a visszaadott index nulla vagy pozitív lesz, ezért a –1 érték jól használható az extra esetek jelzésére. A későbbiekben, amikor majd a mintakereső program fejlettebb változatát írjuk, csak az `strindex` függvényt cseréljük le egy intelligensebbre, és a program többi részét változatlanul hagyjuk. (A standard könyvtár tartalmazza az `strstr` függvényt, amelynek feladata hasonló az `strindex` függvényéhez, de nem a kezdőpont indexét, hanem egy mutatót ad értékül.)

A kezdeti programtervezés után már viszonylag gyorsan megírhatjuk a részletes programot. A szerkezetet követve jól látható az egyes részek egymáshoz kapcsolódása. Egyelőre ne a legáltalánosabb esettel kezdjük: a keresett minta literálisként megadott karaktersorozat legyen. Hamarosan visszatérünk még a karaktertömbök inicializálására és az 5. fejezetben megmutatjuk, hogyan tehetjük a keresett mintát a program futása közben megadható paraméterré. A mintakereső programban megadjuk a `getline` függvény egy újabb változatát. Célszerű, ha ezt összehasonlítjuk az 1. fejezetben leírt változattal. Ezután lássuk a programot!

```
#include <stdio.h>
#define MAXSOR 1000 /* a sor maximális hossza */

int getline(char sor[ ], int max);
int strindex(char forras[ ], char keresett[ ]);
char minta[] = "dal"; /* a keresett minta */
```

```
/* megkeresi a mintát tartalmazó összes sort */
main( )
{
    char sor[MAXSOR];
    int talalt = 0;

    while (getline(sor, MAXSOR) > 0)
        if (strindex(sor, minta) >= 0) {
            printf("%s", sor);
            talalt++;
        }
    return talalt;
}
/* getline: egy sort beolvas s-be és megadja a hosszát */
int getline(char s[ ], int hatar)
{
    int c, i;

    i = 0;
    while (--hatar > 0 && (c = getchar( )) != EOF
           && c != '\n')
                s[i++] = c;
    if (c == '\n')
    s[i++] = c;

    s[i] = '\0';
    return i;
}

/* strindex: visszaadja t indexét s-ben, ill. -1-et,
ha a keresett minta nincs a sorban */
int strindex(char s[ ], char t[ ])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j = i, k = 0; t[k] != '\0' && s[j] == t[k];
               j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

A programban használt összes függvény definíciója

```
visszatérési-típus függvénynév (argumentum deklarációk)
{
    deklarációk és utasítások
}
```

alakú, és a definícióból egyes részek hiányozhatnak. A legegyszerűbb és legrövidebb függvény a

```
dummy( ) { }
```

amely nem csinál semmit és nem ad vissza értéket. Az ilyen üres (semmit nem csináló) függvény beépítése a programba gyakran hasznos, ha a későbbi programfejlesztéshez le akarjuk foglalni egy függvény helyét. Ha a definícióból hiányzik a visszatérési típus, akkor a rendszer az `int` alapfeltételezéssel él.

A program lényegében nem más, mint változók és függvények definícióinak halmaza. A függvények közötti információcsere a függvény argumentumain és visszatérési értékén, valamint a külső változókon keresztül jön létre. A függvények a program forrásállományában tetszőleges sorrendben helyezkedhetnek el, és a forrásprogram több állományra bontható, de egy függvény nem vágható szét két forrásállományba.

A hívott függvény a hívó függvénynek a `return` utasítással adhat vissza értéket. A `return` utasítást tetszőleges kifejezés követheti. Általános alakja:

```
return kifejezés;
```

Ha szükséges, akkor a *kifejezés* típusa a visszatérési típusra konvertálódik. Gyakran a kifejezést zárójelbe tesszük, de ez opcionális.

A hívó függvénynek jogában áll figyelmen kívül hagyni a visszatérési értéket, sőt nem kötelező a `return` utáni kifejezés sem. Ez utóbbi esetben a hívott függvény nem ad vissza értéket a hívó függvénynek. A vezérlés akkor is érték nélkül tér vissza a hívó függvényhez, ha a végrehajtás kilép a függvényből, elérve a függvény végét jelző jobb oldali kapcsos zárójelet. Az ilyen kilépés nem tilos, de valószínűleg valamilyen hibát jelez, ha a függvény értéket ad vissza az egyik helyről és nem ad értéket a másik helyről történő visszatéréskor. Ha egy függvénynek nincs visszatérési értéke, akkor a (formálisan kapott) visszatérési érték határozatlan („szemét").

Az előbbi mintakereső programunk a `main`-ből való kilépéskor egy állapotjelzést ad, ami a találatok száma. Ezt az értéket a programot hívó környezet tetszőleges célra használhatja.

A több forrásállományba szétosztott C programok fordítási és betöltési folyamata rendszertől függő. Például a UNIX operációs rendszer alatt az 1. fejezetben már említett `cc` paranccsal lehet a C programok fordítását vezérelni. Tegyük fel, hogy a három függvényünk három különböző állományban van, amelyek neve `main.c`, `getline.c` és `strindex.c`. Ekkor a

```
cc main.c getline.c strindex.c
```

parancs lefordítja a három forrásállományt, létrehozva ezzel a `main.o`, `getline.o` és `strindex.o` tárgykódú állományokat, majd ezekből összeszerkeszti az `a.out` nevű végrehajtható programállományt. Ha a fordítás közben hiba volt, pl. a `main.c` állományban, akkor a hiba kijavítása után a `main.c` állomány újra fordítható és összeszerkeszthető a korábban kapott tárgykódú állományokkal. Ez a

```
cc main.c getline.o strindex.o
```

paranccsal érhető el. A `cc` parancs a forráskódú és tárgykódú állományokat a `.c` és `.o` névkiterjesztések alapján különbözteti meg.

4.1. gyakorlat. Írjuk meg az `strindex(s, t)` függvénynek azt a változatát, amely a `t` minta `s`-beli legutolsó előfordulásának indexével, vagy ha `t` nem található meg `s`-ben, akkor –1-gyel tér vissza!

4.2. Nem egész értékkel visszatérő függvények

A korábban bemutatott példaprogramok függvényei egész típusú értékkel vagy érték nélkül (`void` típus) tértek vissza. Mi van akkor, ha a függvénynek más típusú értékkel kell visszatérni? Számos matematikai függvény, mint az `sqrt`, `sin`, `cos` stb. `double` típusú értékkel tér vissza, más speciális függvények esetén más a visszatérési típus. Ennek megvalósítását bemutatandó írjuk meg az `atof(s)` függvényt, amely az `s` karaktertömbben megadott adatot kétszeres pontosságú lebegőpontos számmá alakítja. Az `atof` a 2. és a 3. fejezetben, különböző változatokban bemutatott `atoi` függvény kiterjesztése: lehetővé teszi az opcionálisan megadott előjel és tizedespont kezelését, valamint az egész, ill. törtrész megléte vagy hiánya esetén is használható. Mindezek ellenére ez a program nem egy minden igényt kielégítő bemeneti konverziós eljárás, egy ilyen eljárás megírása több helyet igényelne. A standard könyvtár tartalmaz egy `atof` függvényt az `<stdlib.h>` headerben.

```
#include <ctype.h>

/* atof: az s karaktersorozat duplapontos számmá alakítása */
double atof(char s[ ])
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++)
        /* az üres helyek átugrása */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
        if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}
```

Először is az `atof` függvénynek deklarálnia kell a visszatérési típusát, mivel az most nem `int`. A visszatérési típus megadása a függvény neve előtt történik. Másodszor, és ez legalább ilyen fontos, a hívó eljárással is tudatni kell, hogy az `atof` visszatérési értéke nem `int` típusú. Ennek egyik módja, hogy az `atof` függvényt explicit módon deklaráljuk a hívó eljárásban. A deklaráció módját a következő primitív kalkulátorprogramon keresztül mutatjuk be. A program soronként egy-egy számot olvas be, amely előtt előjel is lehet, majd a számokat összeadja és az eredményt minden adatbeolvasás után kiírja.

```
#include <stdio.h>
```

```
#define MAXSOR 100
/* primitív kalkulátorprogram */
main( )
{
    double sum, atof(char [ ]);
    char sor[MAXSOR];
    int getline(char sor[ ], int max);

    sum = 0;

    while (getline(sor, MAXSOR) > 0)
        printf("\t%g\n", sum += atof(sor));
    return 0;
}
```

A

```
double sum, atof(char[ ]);
```

deklaráció azt mondja ki, hogy a `sum` egy `double` típusú változó, valamint az `atof` függvénynek egyetlen, `char[ ]` típusú argumentuma van és visszatérési értéke is `double` típusú.

Az `atof` függvényt következetesen, egymással összhangban kell deklarálni és definiálni. Ha egyetlen forrásállományon belül ellentmondás van az `atof` típusa és a `main`-beli hívásának típusa között, akkor ezt a hibát a fordítóprogram észreveszi és jelzi. De ha az `atof` függvényt önállóan fordítjuk le (és legtöbbször így van), akkor a hiba nem derül ki. Az `atof` visszatér a `double` típusú eredménnyel, amit a `main` `int` típusúként kezel és ez értelmetlen eredményre vezet.

Az elmondottak alapján látszik, hogy a deklaráció és a definíció összhangjára vonatkozó szabályt komolyan kell venni. A típusillesztési hiba főképp olyankor fordul elő, ha nincs függvényprototípus, és a függvény a kifejezésbeli első előfordulásakor, implicit módon van deklarálva. Ilyen pl. a

```
sor += atof(sor)
```

kifejezés. Ha egy korábban még nem deklarált név fordul elő egy kifejezésben és a nevet bal oldali kerek zárójel követi, akkor arról a rendszer a programkörnyezet alapján feltételezi, hogy függvény és a kifejezés megfelelő részét a függvény deklarációjának tekinti. A fordítóprogram ugyancsak feltételezi, hogy az így deklarált függvény egész típusú (`int`) visszatérési értéket ad, viszont semmit sem tételez fel az argumentumairól. Abban az esetben, ha a függvénydeklarációban nincs argumentum, mint pl. a

```
double atof( );
```

deklarációban, akkor a fordítóprogram ismét nem tételez fel semmit az argumentumokról és a teljes paraméter-ellenőrzést kikapcsolja. Ennek az a célja, hogy az üres paraméterlistájú deklarációkat tartalmazó régebbi programok is lefordíthatók legyenek az új fordítóprogramokkal. Mindezek ellenére az üres paraméterlista nyújtotta lehetőségeket ne alkalmazzuk az új programokban.

Ha adott a megfelelően deklarált `atof` függvény, akkor azt felhasználva egyszerűen megírhatjuk az `atoi` függvényt is (amely egy karaktersorozatot `int` típusú számmá alakít):

```
/* atoi: az s karaktersorozat egésszé alakítása */
int atoi(char s[ ])
{
    double atof(char s[ ]);

    return (int) atof(s);
}
```

Figyeljük meg a deklarációk szerkezetét és a return utasítást! A

```
return kifejezés;
```

utasításban a kifejezés értéke a visszatéréskor automatikusan `int` típusúvá konvertálódik, ami az `atoi` típusa. Így az automatikus konverzió a double típusú `atof` értéke esetén is létrejön, ha az megjelenik a `return` utasításban. Ezért a visszatéréskor megadott `int` típuskijelölés felesleges, bár a fordítóprogram figyeli és felhasználja. A kényszerített típuskijelölést csak azért használtuk, hogy elkerüljük a fordítóprogram bármiféle figyelmeztető jelzését.

4.2. gyakorlat. Bővítsük ki az `atof` függvényt úgy, hogy az pl. az `123.45e-6` alakú tudományos jelölésmódot is kezelni tudja! A bemeneti karaktersorozat a kitevő jelzésére `e` vagy `E` karaktereket használhatja és utána előjeles kitevő következhet.

4.3. A külső változók

A C nyelvű program külső objektumok – változók vagy függvények – halmaza. A külső jelzőt a belső ellentéteként használjuk, ami a függvények belsejében definiált argumentumok és változók leírására alkalmas. A külső változókat a függvényen kívül definiáljuk, így elvileg több függvényben is felhasználhatók. A függvények maguk mindig külső típusúak, mivel a C nyelv nem engedi meg, hogy egy függvény belsejében újabb függvényt definiáljunk. Alapértelmezés szerint a külső változókra vagy függvényekre azonos néven való hivatkozás (még akkor is, ha külön fordított függvényből történik) mindig azonos dolgot jelent (a szabvány ezt a tulajdonságot *külső csatolásnak, external linkage*-nek nevezi). Ilyen értelemben a C külső változói analógak a FORTRAN COMMON változóival vagy a Pascal programok legkülső blokkjában használt változókkal. A későbbiekben majd látni fogjuk, hogy hogyan lehet olyan külső változókat és függvényeket definiálni, amelyek csak egyetlen forrásállományban láthatók.

Mivel a külső változók globálisan hozzáférhetők, jól használhatók a függvények argumentumai és visszatérési értékei helyett, azaz a függvények közti adatforgalomban. Bármely függvény hozzáférhet egy külső változóhoz a neve alapján, ha az valahol ezen a néven deklarálva volt.

Ha a függvények közötti adatforgalomhoz sok változó kell, akkor a külső változók használata kényelmesebb és hatékonyabb, mint a hosszú argumentumlista. De ahogy már az 1. fejezetben is említettük, ezt a megállapítást fenntartással kell fogadnunk, mivel a sok külső változó rontja a program áttekinthetőségét, és túl sok (sokszor nem kívánt) adatkapcsolathoz vezet.

A külső változók a nagyobb hatáskörük és élettartamuk miatt is hasznosak. Az automatikus változók csak a függvény belsejében léteznek: létrejönnek a függvénybe való belépéskor és megszűnnek a kilépéskor. A külső változók állandóak, értékük az egyik függvényhívástól a másikig megmarad. Így ha két függvény azonos adathalmazzal dolgozik

(de egyik sem hívja a másikat), akkor kényelmesebb a közös adathalmazt külső változóként használni az argumentumlistán keresztüli körülményes adatátadás helyett.

Az eddigi gondolatokat vizsgáljuk meg egy nagyobb példán keresztül. A feladat egy kalkulátorprogram írása, amely már végrehajtja a +, –, *, / alapműveleteket is. Az egyszerű megvalósíthatóság miatt a kalkulátor használja a fordított lengyel jelölésmódot a szokásos infix jelölésmód helyett. (Néhány zsebszámológép, valamint a Forth és Postcript programnyelvek is a fordított lengyel jelölésmódot használják.)

A fordított lengyel jelölésmódban az operátor az operandusok után következik. Az infix formában írt

$$(1 - 2) * (4 + 5)$$

kifejezés fordított lengyel jelölésmódban

$$1\ 2\ -\ 4\ 5\ +\ *$$

tehát így kell majd a programnak beadni. A zárójelekre nincs szükség, a jelölésmód teljesen egyértelmű mindaddig, amíg tudjuk, hogy az egyes operátorok hány operandust várnak.

A fordított lengyel jelölésmódot feldolgozó program megvalósítása nagyon egyszerű: az egyes operandusokat egy verembe tesszük, majd ha megérkezett az operátor, akkor a megfelelő számú (általában kettő) operandust kivesszük a veremből, alkalmazzuk rájuk az operátort, majd az eredményt visszatesszük a verembe. A fenti példában először 1, majd 2 kerül a verembe, amiből az első az operátor megérkezése után helyettesítődik az eredménnyel (–1-gyel). Ezután a 4 és 5 kerül a verembe, amit az összegükkel (9-cel) helyettesítünk. Most a veremben –1 és 9 van, amit a szorzatukkal helyettesítünk. A műveletsor befejeztével a verem tetején lévő értéket kivesszük és kiírjuk. A műveletsor a bemenetre adott sor végét elérve zárul.

A program lényegében egy ciklusból áll, amely végrehajtja az egyes operátorokkal és operandusokkal a megfelelő műveleteket. A program váza:

```
while (a következő operandus vagy operátor nem állományvége-jel)
    if (szám)
        tedd a verembe
    else if (operátor)
        vedd elő az operandusokat
        hajtsd végre a műveletet
        tedd az eredményt a verembe
    else if (újsor-jel)
        vedd ki a verem tetején lévő elemet és írd ki
    else
        hiba
```

Egy adat verembe helyezése (push) és kivétele (pop) nagyon egyszerű művelet, de a hibafigyeléssel és hiba utáni helyreállítással már viszonylag hosszú programrészt ad, amit a programon belül többször ismételni kellene, ezért inkább önálló függvényként valósítottuk meg azokat. Szintén önálló függvény a következő bemeneti operátor vagy operandus beolvasását végző rész.

Egy fontos programtervezési kérdésről még nem döntöttünk: hol legyen a verem és melyik eljárások kezelhetik közvetlenül a vermet. Egy lehetőség, hogy a verem a `main` függvényben van, és az adatokat a verembe író, ill. onnan kiolvasó eljárások-

nak paraméterként átadjuk a vermet, ill. az aktuális veremmutatót. A `main` függvénynek nem kell tudni a vermet vezérlő változókról, csak a verembe írást, ill. az onnan való olvasást kell vezérelnie. Ezért úgy döntöttünk, hogy a verem és minden vele kapcsolatos információ legyen külső változó, amelyhez csak a `push` és `pop` függvény férhet hozzá, a `main` nem.

Ezt az elgondolást utasítások formájában egyszerűen felírhatjuk. Arra gondolva, hogy a program egyetlen forrásállományban van, a szerkezete valahogy így fog kinézni:

```
#include-ok
#define-ok
main függvény deklarációi
main( ) { ... }

külső változók a push és pop számára

void push(double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }

a getop függvény által hívott eljárások
```

A későbbiekben majd megmutatjuk, hogy a program hogyan osztható két vagy több forrásállományra.

A `main` függvény ciklusa egy nagy `switch` utasítást tartalmaz, amely az operandusok és operátorok jellege szerint választja szét a feladatokat. Ez talán tipikusabb alkalmazása a `switch` utasításnak, mint a 3.4. pontban bemutatott példa.

```
#include <stdio.h>
#include <stdlib.h> /* az atof() miatt */

#define MAXOP 100 /* az operandus vagy operátor
                     max. hossza */
#define SZAM '0' /* jelzi, hogy számot talált */

int getop(char[ ]);
void push(double);
double pop(void);

/*fordított lengyel jelölésmóddal működő kalkulátorprogram */

main( )
{
    int tipus;
    double op2;
    char s[MAXOP];

    while ((tipus = getop(s)) != EOF) {
        switch (tipus) {
        case SZAM:
            push(atof(s));
            break;
```

```
            case '+':
                push(pop( ) + pop( ));
                break;
            case '*':
                push(pop( ) * pop( ));
                break;
            case '-':
                op2 = pop( );
                push(pop( ) - op2);
                break;
            case '/':
                op2 = pop( );
                if (op2 != 0.0)
                    push(pop( ) / op2);
                else
                    printf("Hiba: osztás nullával\n");
                break;
            case '\n':
                printf("\t%.8g\n", pop( ));
                break;
            default:
                printf("Hiba: ismeretlen parancs %s\n", s);
                break;
            }
        }
        return 0;
    }
```

Mivel a + és * kommutatív operátorok, ezért mindegy, hogy a veremből kivett operandusokat milyen sorrendben dolgozzuk fel, viszont a – és / esetén a bal és jobb oldali operandusok nem cserélhetők fel. Ezért a

```
push(pop( ) - pop( )); /* HIBÁS! */
```

utasításban a pop hívási sorrendek definiálatlanok lehetnek. A művelet helyes sorrendben való végrehajtása érdekében a veremből elsőnek kivett értéket egy átmeneti tárolóba kell helyezni, mint a main is teszi. A verembe írást, ill. az onnan való olvasást végző függvények:

```
#define MAXVAL 100 /* a val tömbbel kialakított verem max. mélysége */
int sp = 0; /* a verem következő szabad helye */
double val[MAXVAL]; /* a verem tömbje */

/* push: f értékét a verembe teszi */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("Hiba: a verem megtelt, nem írható ki %g\n", f);
}
```

```
/* pop: kiolvas egy adatot a veremből és visszatér az értékkel */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("Hiba: a verem üres\n");
        return 0.0;
    }
}
```

Egy változó külső tárolási osztályú, ha bármelyik függvényen kívül definiáljuk. Így a verem és a verem indexe, amelyet a push és a pop közösen használ, ezeken függvényeken kívül lett definiálva. Mivel a definiálás a push és a pop függvényekkel együtt történt, a verem változóihoz a main közvetlenül nem fér hozzá, azok a számára láthatatlanok.
Most nézzük a getop függvényt, amelynek feladata a következő operátor vagy operandus előkészítése. A megoldás egyszerű: olvasni kell a bemeneti karaktersorozatot és a szóközöket, ill. tabulátorokat át kell ugrani. Ha a következő értékes karakter nem számjegy vagy tizedespont, akkor annak értékével vissza kell térni a hívó függvénybe, egyébként pedig össze kell gyűjteni a számjegyeket (amelyek között előfordulhat a tizedespont is) és vissza kell térni a SZAM értékkel, ami jelzi, hogy egy szám van összegyűjtve. A teljes getop függvény:

```
#include <ctype.h>
int getch(void);
void ungetch(int);

/* getop: megadja a következő operátort
vagy számot (operandust) */
int getop(char s[ ])
{
    int i, c;

    while ((s[0] = c = getch( )) == ' ' ||
    c == '\t')

        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c; /* nem szám */
    i = 0;
    if (isdigit(c)) /* összegyűjti az egész részt */
        while (isdigit(s[++i] = c = getch( )))
            ;
    if (c == '.') /* összegyűjti a törtrészt */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return SZAM;
}
```

Mit csinál a getch és ungetch függvény? Gyakran a programban nem lehet megállapítani, hogy mikor olvastunk eleget a bemenetről, csak ha már túl sokat olvastunk. Itt egy ilyen eset, amikor a számokat gyűjtjük össze: amíg az első nem szám karaktert meg nem találtuk, addig a szám még nem teljes. De amikor a program beolvassa az első nem szám karaktert, akkor már túlfutott az olvasással, erre a karakterre még nincs felkészülve.

A probléma megoldható, ha lehetőségünk van a feleslegesen beolvasott karaktert „nem beolvasottá" tenni. Így, ha a program bármikor egy karakterrel többet olvasott, akkor ezt a karaktert „visszateheti" és a fennmaradó karaktersorozat úgy viselkedik, mintha ezt a karaktert soha nem olvastuk volna be. Szerencsére a „nem beolvasottá" tétel egyszerűen szimulálható egy együttműködő függvénypárral. A getch fogja szolgáltatni a vizsgálathoz a következő bejövő karaktert, az ungetch pedig elvégzi a karakter „visszaírását" a bemenetre, így a következő getch hívás ismét ezt a karaktert veszi elő.

A két függvény együttműködése nagyon egyszerű: az ungetch a „visszaírandó" karaktert a két függvény által közösen használt pufferba (karakteres tömbbe) helyezi. A getch ebből a pufferből olvas, ha van benne valami és hívja a getchar függvényt, ha a puffer üres. Az aktuális karakter pufferbeli helye egy indexváltozóval adható meg.

Mivel a puffert és az indexváltozót a getch és az ungetch közösen használja és értéküket a két hívás között is meg kell tartani, ezért a két függvényre nézve külső változók. A getch és ungetch függvények, valamint a közös változók:

```
#define BUFSIZE 100

char buf[BUFSIZE]; /* az ungetch puffere */
int bufp = 0;
/* a puffer következő szabad helye */
int getch(void)    /* a következő (esetleg korábban
                      visszaírt) karakter bevétele */
{
    return (bufp > 0) ? buf[--bufp] : getchar( );
}
void ungetch(int c) /* visszaír egy karaktert a bemenetre */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: puffer túlcsordulás\n");
    else
        buf[bufp++] = c;
}
```

A standard könyvtárban is található egy ungetc nevű függvény, amely egyetlen karaktert „ír vissza" a bemenetre. Erre a 7. fejezetben még visszatérünk. A példánkban egy karakteres puffer helyett egy tömböt használtunk, ami sokkal általánosabb megközelítése a problémának.

4.3. gyakorlat. Adott a kalkulátorprogram váza. Bővítsük ezt ki a modulus (%) operátorral és gondoskodjunk a negatív számok (egyoperandusú -) kezeléséről!

4.4. gyakorlat. Bővítsük a programot új parancsokkal! Az egyik parancs írja ki a verem tetején lévő elemet anélkül, hogy az a veremből eltűnne, a másik cserélje meg a verem tetején lévő két elemet, a harmadik készítsen másolatot a verem tetején lévő elemről, a negyedik pedig törölje a vermet.

4.5. gyakorlat. Tegyük lehetővé, hogy a kalkulátorprogramunk hozzáférjen olyan könyvtári függvényekhez, mint `sin, exp` és `pow`. Ezek a függvények a `<math.h>` headerben vannak, aminek a leírása a B. Függelék 4. pontjában található.

4.6. gyakorlat. Bővítsük úgy a programot, hogy képes legyen változók kezelésére is. (Ezt könnyű megvalósítani, ha 26 változót engedünk meg, és minden változó nevéül az angol ábécé egy betűjét választjuk.) Rendeljünk egy változót a legutoljára kiírt értékhez is.

4.7. gyakorlat. Írjunk `ungets(s)` néven függvényt, amely egy teljes karaktersorozatot „visszaír" a bemenetre! Az `ungets` függvény kezelje közvetlenül a `buf` és `bufp` változókat, vagy egyszerűen csak használja az `ungetch` függvényt?

4.8. gyakorlat. Tegyük fel, hogy soha nem akarunk egynél több karaktert „visszaírni" a bemenetre. Módosítsuk ennek megfelelően a `getch` és `ungetch` függvényeket!

4.9. gyakorlat. A példában használt `getch` és `ungetch` függvények nem kezelik helyesen a „visszaírt" `EOF` karaktert. Határozzuk meg a helyes `EOF` kezelés módját és egészítsük ki ezzel a programtervet!

4.10. gyakorlat. Tegyük fel, hogy egy `getline` függvénnyel a teljes bemeneti sort egyszerre olvassuk be. Ekkor nincs szükség a `getch` és `ungetch` függvényekre. Gondoljuk át, hogy ez hogyan módosítja a kalkulátorprogramot!

4.4. Az érvényességi tartomány szabályai

A C nyelvű programban szereplő összes függvényt és külső változó definíciót nem szükséges egyszerre fordítani, a forrásprogram több független állományban tartható és a korábban már lefordított részek a könyvtárból betölthetők. Ezzel kapcsolatban a következő fontos kérdések merülnek fel:

- Hogyan lehet a deklarációkat úgy megírni, hogy a változók a fordítás során megfelelően deklaráltak legyenek?
- Hogyan kell a deklarációkat elrendezni ahhoz, hogy betöltéskor a program minden része megfelelően kapcsolódjon egymáshoz?
- Hogyan szervezzük meg a deklarációkat, hogy mindegyik csak egyszer forduljon elő?
- Hogyan lehet a külső változókat inicializálni?

A kérdések megválaszolásához szervezzük át a kalkulátorprogramunkat úgy, hogy az több forrásállományban legyen. Gyakorlatilag a kalkulátorprogram túl kicsi ahhoz, hogy értelmesen részekre bontsuk, de mégis jól mutatja a teendőket nagyobb programok esetén.

Egy név érvényességi tartománya (hatásköre) az a programrész, amiben a nevet használhatjuk. A függvény kezdetén deklarált automatikus változó érvényességi tartománya az a függvény, amelyben deklarálták. A helyi (lokális) változók neve más függvényben ismeretlen. Ugyanez igaz a függvények paramétereire is, mivel ezek valójában helyi változók.

A külső változók vagy függvények érvényességi tartománya a deklaráció helyén kezdődik és az éppen fordított forrásállomány végéig tart. Például, ha a `main, sp, val, push` és `pop` egy állományban van definiálva az előbbi sorrendben, vagyis

```
main( ) { ... }

int sp = 0;
double val[MAXVAL];
```

```
void push(double f) { ... }

double pop(void) { ... }
```

akkor az `sp` és `val` változók egyszerűen, a nevük megadásával használhatók a `push` és `pop` függvényekben, külön deklarációra nincs szükség. Ugyanakkor ezek a nevek (a `push` és `pop` függvénynevekkel együtt) ismeretlenek a `main` számára.

Másrészről, ha egy külső változóra a definiálása előtt hivatkozunk, vagy más forrásállományban definiáltuk, mint ahol használjuk, akkor kötelező az `extern` deklaráció.

Fontos, hogy megkülönböztessük a külső változók *deklarálását* és *definiálását*. A deklaráció a változó tulajdonságait (elsősorban a típusát) írja le, a definíció viszont ezen-kívül még tárterületet is rendel hozzá. Ha az

```
int sp;
double val[MAXVAL];
```

sorok bármely függvényen kívül jelennek meg, akkor *definiálják* az `sp` és `val` külső változókat, tárterületet rendelnek hozzájuk és a forrásállomány további része számára deklarációként is működnek. Másrészt az

```
extern int sp;
extern double val[ ];
```

sorok a forrásállomány további része számára *deklarálják*, hogy `sp` `int` típusú és hogy `val` `double` típusú tömb (amelynek méretét máshol adjuk meg), de nem rendelnek tárterületet ezekhez a változókhoz.

A külső változókat csak a forrásállományok egyikében kell *definiálni*, a többi állományban csak `extern` deklaráció van, amelyen keresztül ezek a változók elérhetők. (A definíciót tartalmazó állományban is lehet `extern` deklaráció.) A tömbök méretét a definícióban kell megadni, de opcionálisan szerepelhet az `extern` deklarációban is.

A külső változók inicializálása csak a definícióval együtt történhet.

Bár nem nagyon valószínű, de tegyük fel, hogy a `push` és `pop` függvényeket az egyik forrásállományban definiáltuk és inicializáltuk. Ekkor az összekapcsolásukhoz az alábbi definíciók és deklarációk szükségesek:

Az 1. állományban:

```
extern int sp;

extern double val[ ];

void push(double f) { ... }

double pop(void) { ... }
```

A 2. állományban:

```
int sp = 0;
double val[MAXVAL];
```

Mivel az *1. állományban* az `extern` deklarációk a függvénydefiníciókon kívül van-

nak, ezért a teljes *1. állomány* számára elegendő ez a deklaráció. Ugyanez a szervezés szükséges akkor is, ha az `sp` és `val` definícióját egy állományban megelőzi a rájuk való hivatkozás.

4.5. A header állományok

Most ismét foglalkozzunk a kalkulátorprogramunk több állományba osztásával, mintha az egyes részek sokkal nagyobbak lennének. A `main` függvény menne a `main.c` nevű állományba; a `push, pop` és a változóik egy másik, `stack.c` nevű állományba; a `getop` a `getop.c` nevű állományba; végül a `getch` és `ungetch` a negyedik, `getch.c` nevű

`calc.h`

```
#define SZAM '0'
void push(double);
double pop(void);
int getop(char[]);
int getch(void);
void ungetch(int);
```

`main.c`

```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main( ) {
    ...
}
```

`getop.c`

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop( ) {
    ...
}
```

`stack.c`

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

`getch.c`

```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
int bufp = 0;
int getch(void) {
...
}
void ungetch(int) {
...
}
```

állományba. Azért választottuk szét az egyes részeket egymástól, mert egy tényleges programban is külön lefordított könyvtárakat használunk.

Most már csak egy nehézséget kell megoldanunk: a definíciók és deklarációk szétosztását az egyes állományok között. Amennyire csak lehetséges, igyekszünk a definíciókat és deklarációkat centralizálni, hogy csak egy példányt kelljen karbantartani és figyelemmel kísérni. Ennek megfelelően ezt a közös definíciós–deklarációs részt egy `calc.h` nevű *header állományba* helyezzük és szükség esetén `include` utasítással hozzáfűzzük a programhoz. (A `#include` utasítást a 4.11. pontban tárgyaljuk részletesen.) Az így létrejövő programszerkezet látható a 96. oldalon.

Az a kívánság, hogy minden egyes programrész csak a feladatához szükséges információkhoz férjen hozzá, valamint a gyakorlati megvalósíthatóság között kompromisszumos döntést kell hozni, de mindenesetre több header állomány kézbentartása nagyon nehéz feladat. Közepes programméretekig valószínűleg a legjobb megoldás, ha egyetlen header állományba foglalunk mindent, ami az egyes programrészek együttműködéséhez szükséges, és csak nagyon nagy programoknál alkalmazunk bonyolultabb szervezést és több header állományt.

4.6. A statikus változók

A `stack.c` állomány `sp` és `val` változói, valamint a `getch.c` állomány `buf` és `bufp` változói az egyes forrásállományokban lévő függvények saját változói, és bárhol máshol nem hozzáférhetők. A külső változókra vagy függvényekre alkalmazott `static` deklaráció az objektum érvényességi tartományát az éppen fordított forrásállomány fennmaradó részére korlátozza. Így a külső változók `static` deklarálása jó lehetőséget nyújt az `ungetch-getch` függvénypár `buf` és `bufp` változóihoz hasonló változók (amelyeknek a közös használat miatt külső változóknak kell lenni) más függvények vagy programrészek előli elrejtésére (pl. a fenti példában a `getch` vagy `ungetch` felhasználói nem látják a `buf` vagy `bufp` változókat, bár azok külső változók).

A statikus tárolási osztály a normális deklaráció elé írt `static` szóval deklarálható. Ha, mint a következő példában, a két változó és a két függvény egyetlen forrásállományból lesz lefordítva, akkor semmilyen más függvény nem férhet a `buf` és `bufp` változókhoz, és neveik ugyanezen program más forrásállományaiban szabadon használhatók.

```
static char buf [BUFSIZE];    /* az ungetch puffere */
static int bufp = 0;      /*a következő szabad hely a
                            pufferban */
int getch(void) { ... }
void ungetch(int c) { ... }
```

A `push` és `pop` veremkezelési műveletei ugyanígy elrejthetők, ha az `sp` és `val` változókat `static` típusúnak deklaráljuk.

A külső statikus deklarációt leggyakrabban változókra alkalmazzák, de függvényekre is használható. Normális körülmények közt a függvények nevei globálisak, a teljes program számára ismertek. Ha egy függvényt `static` tárolási osztályúnak deklarálunk, akkor a neve a deklarációt tartalmazó forrásállományon kívül nem ismert.

A statikus tárolási osztály a belső változókra is alkalmazható. A belső statikus változók a megfelelő függvényre nézve lokálisak, csakúgy, mint az automatikus változók, de ellentétben azokkal állandóan megmaradnak (az automatikus változók a függvény hívásakor jönnek létre és a függvényből visszatérve megszűnnek). Ez azt jelenti, hogy a belső `static` deklarálású változók a függvény saját, állandó tárolóhelyei lehetnek.

4.11. gyakorlat. Módosítsuk a `getop` függvényt úgy, hogy ne kelljen használnia az `ungetch` függvényt! Segítség: használjunk belső statikus változót!

4.7. Regisztervváltozók

A `register` deklaráció azt tudatja a fordítóprogrammal, hogy az így deklarált változót nagyon gyakran fogjuk használni. Az elképzelés az, hogy a `register` deklarálású változót a számítógép regiszterébe helyezzük, ami kisebb méretű és gyorsabb programot eredményez. A fordítóprogramnak lehetősége van figyelmen kívül hagyni a deklarációt. A `register` deklaráció általános alakja:

```
register int x;
register char c;
```

A `register` tárolási osztály csak automatikus változókra és függvények formális paramétereire írható elő. Ez utóbbi eset

```
f(register unsigned m, register long n)
{
    register int i;
    ...
}
```

módon valósítható meg.

A gyakorlatban a regisztervváltozókra az alkalmazott hardver miatt megszorítások érvényesek. Egy függvényen belül csak néhány változó lehet regiszteres és csak bizonyos típusú változók. A felesleges `register` deklarációk nem okoznak problémát, mivel a felesleges számú vagy nem megengedett típusú változók deklarálásából a `register` szó törlődik. További megszorítás, hogy nem hivatkozhatunk a regisztervváltozó címére (ezzel a kérdéssel az 5. fejezetben még foglalkozunk), függetlenül attól, hogy aktuálisan egy regiszterben helyezkedik-e el vagy sem. A regisztervváltozókra vonatkozó specialitások és korlátozások gépről gépre változnak.

4.8. Blokkstruktúra

A Pascalhoz vagy hasonló nyelvekhez viszonyítva a C nyelv nem egy blokkstrukturált nyelv, mivel a függvények nem definiálhatók más függvények belsejében. Másrészről viszont a függvények belsejében a változók blokkstrukturált módon deklarálhatók. A változó deklarációja (és vele együtt az inicializálása) *bármelyik* összetett utasítást kezdő bal oldali kapcsos zárójel után következhet, nem csak a függvény kezdetén. Az így deklarált változók rejtve maradnak a külső blokkok azonos nevű változói elől, és csak addig léteznek, amíg a vezérlés el nem jut a blokk záró, jobb oldali kapcsos zárójeléig. Például az

```
if (n > 0) {
    int i; /* itt új i változót deklarálunk */

    for (i = 0; i < n; i++)
        ...
}
```

programrészben az i változó érvényességi tartománya az if utasítás igaz feltételhez tartozó ága, és nincs semmiféle kapcsolata a blokkon kívüli i változóval. Egy adott blokkban a deklarációval együtt inicializált automatikus változó a blokkba való minden belépéskor újra inicializálódik. A static tárolási osztályúnak deklarált változó csak a blokkba való első belépéskor inicializálódik.

Az automatikus változók (beleértve a formális paramétereket is) szintén rejtve maradnak az azonos nevű külső változók és függvények elől. Nézzük a következő deklarációkat:

```
int x;
int y;

f(double x)
{
    double y;
    ...
}
```

A függvény belsejében x-re paraméterként hivatkozhatunk és double típusú, ezzel szemben a függvényen kívül külső tárolási osztályú, int típusú változó. Ugyanez igaz az y-ra is.

A bemutatott példák ellenére érdemesebb elkerülni az azonos nevű, eltérő érvényességi tartományú változónevek használatát, mivel túl nagy a hibázás lehetősége.

4.9. Változók inicializálása

Az inicializálásról már többször beszéltünk, de mindig csak érintőlegesen, más téma kapcsán. Ebben a pontban összegezzük az inicializálás szabályait, figyelembe véve a tárolási osztályokról eddig elmondottakat.

Explicit inicializálás hiányában a külső és statikus változók kezdeti értéke garantáltan nulla lesz, az automatikus és regisztervváltozók kezdeti értéke viszont határozatlan.

Skaláris változók a definíciójukkal együtt inicializálhatók, a nevüket követő egyenlőségjel után írt kifejezéssel. Például:

```
int x = 1;

char aposztrof = '\'';
long nap = 1000L * 60L * 60L * 24L;
     /* egy nap hossza ms-ban */
```

Külső és statikus változók esetén a kezdeti érték csak állandó kifejezéssel adható meg és az inicializálás csak egyszer, a program végrehajtásának kezdete előtt jön létre. Automatikus és regisztervváltozók esetén az inicializálás minden alkalommal megtörténik, amikor a vezérlés a függvényre vagy blokkra kerül.

Automatikus vagy regisztervváltozók esetén a kezdeti érték nem csak állandó lehet, kezdeti értékként megengedett bármilyen, korábban definiált értékű változót vagy függvényhívást tartalmazó kifejezés is. Például a 3.3. pontban leírt bináris kereső program inicializáló része

```
int binsearch(int x, int v[ ], int n)
{
    int also = 0;
    int felso = n - 1;
    int kozep;
        ...
}
```

alakban is írható az ott látott

```
int also, felso, kozep;

also = 0;
felso = n - 1;
```

alak helyett. Az automatikus változók ilyen inicializálása lényegében az értékadó utasítás rövidítéseként fogható fel. Ízlés dolga, hogy az inicializálás melyik alakját részesítjük előnyben. A könyvben általában az explicit értékadást használjuk, mivel a deklarációban elhelyezett kezdeti érték nehezebben vehető észre, a program pedig nehezebben követhető.

Tömbök szintén inicializálhatók a deklarációjukkal együtt, a deklarációt követő egyenlőségjel után kapcsos zárójelbe írt kezdetiérték-listával. A lista egyes elemeit vessző választja el. Például a hónapokban lévő napok számát tartalmazó `napok` tömb az

```
int napok[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

módon inicializálható. Ha a tömb mérete a deklarációból hiányzik, akkor a fordítóprogram a kezdeti értékek leszámolásával meghatározza a tömb hosszát (a fenti példában a tömb hossza 12).

Ha a tömb deklarált méreténél kevesebb kezdeti értéket adunk meg, akkor a külső, statikus és automatikus tárolási osztály esetén a hiányzó elemek nulla kezdeti értéket kapnak. Hibát csak az okoz, ha a kezdeti értékek száma nagyobb, mint a tömb mérete. Más nyelvekkel ellentétben nincs mód arra, hogy egy kezdeti értéket ismétlési tényezővel több elemhez is hozzárendeljünk, vagy hogy egy tömb közbenső elemének kezdeti értéket rendeljünk anélkül, hogy a többi elem értéket kapna.

A karaktertömbök inicializálása speciális módon megy végbe: a hozzárendelt karaktersorozat kapcsos zárójelek és elválasztó vesszők nélkül adható meg. Például:

```
char minta[ ] = "dal";
```

adható meg, ami a vele egyenértékű

```
char minta[ ] = {'d', 'a', 'l', '\0'};
```

alak rövidebb változata. Ebben az esetben a tömbnek négy eleme van, a három karakter és a `'\0'` végjelzés.

4.10. Rekurzió

A C nyelv függvényei rekurzívan használhatók: egy függvény közvetlenül vagy közvetetten hívhatja saját magát. Vizsgáljuk meg a számot, mint karaktersorozatot kiíró

programunkat. Ahogy elmondtuk, a számjegyek rossz sorrendben keletkeznek, az alacsonyabb helyiértékű számjegy előbb áll rendelkezésünkre, mint a magasabb helyiértékű, amivel a kiírást kezdeni kellene. A probléma megoldására két lehetőség van. Az egyik, hogy a számjegyeket a keletkezésük sorrendjében egy tömbbe tároljuk, majd fordított sorrendben írjuk ki (így működött a 3.6. pontban bemutatott `itoa` példaprogramunk is). A másik lehetőség egy rekurzív program, amelyben a `printd` függvény először saját magát hívja meg, hogy feldolgozhassa a magasabb helyiértékű számjegyeket, majd utána írja csak ki az utolsó számjegyet. Ez a változat is hibás eredményt adhat a legnagyobb negatív szám esetén. A `printd` függvény programja:

```
#include <stdio.h>

/* printd: n szám kiírása decimális formában */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n/10);
    putchar(n % 10 + '0');
}
```

Amikor egy függvény rekurzívan hívja saját magát, minden híváskor az automatikus változók új készletével kezdi a munkát. Ez az új változókészlet teljesen független a korábbi hívásokkor keletkező készletektől. Így pl. a `printd(123)` esetén a `printd` első hívásakor az `n = 123` argumentumot kapja, amiből `n = 12` argumentumot ad át a második `printd` híváskor és `n = 1` argumentumot a harmadik híváskor. A harmadik híváskor a `printd` kiírja az 1 értéket, visszatér a második hívási szintre, ahol kiírja a 2 értéket, végül visszatérve az első hívási szintre kiíródik a 3 érték és a folyamat befejeződik.

Egy másik jó példa a rekurzióra a quicksort rendező algoritmus, amit C. A. R. Hoare 1962-ben dolgozott ki. Az algoritmus lényege, hogy adott egy tömb, amelynek egy elemét kiválasztjuk, a többi elemet pedig két részhalmazra osztjuk úgy, hogy az egyikbe a kiválasztott elemnél nagyobb vagy azzal egyenlő, a másikba pedig az annál kisebb elemek kerüljenek. Ezt az eljárást ezután rekurzívan alkalmazzuk a két részhalmazra. Amikor egy részhalmaz kettőnél kevesebb elemet tartalmaz, már nem szükséges tovább rendezni és leállítjuk a rekurziót.

Az itt ismertetett quicksort programunk nem a lehetséges leggyorsabb változat, de mindenesetre az egyik legegyszerűbb. A programban a részhalmazokra (résztömbökre) osztáshoz a középső elemet használjuk.

```
/* qsort: a v[bal] ... v[jobb] tömb rendezése növekvő sorrendben */
void qsort(int v[ ], int bal, int jobb)
{
    int i, utolso;
    void swap(int v[ ], int i, int j);

    if (bal >= jobb) /* semmit nem csinál, ha */
        return; /* kettőnél kevesebb elemből áll */
    swap(v, bal, (bal + jobb)/2); /* a kiválasztott */
```

```
    utolso = bal; /* elemet a v[0] helyre rakja */
    for (i = bal + 1; i <= jobb; i++) /* felbontás */
        if (v[i] < v[bal])
            swap(v, ++utolso, i);
    swap(v, bal, utolso); /* a kiválasztott elem helyretétele */
    qsort(v, bal, utolso-1);
    qsort(v, utolso+1, jobb);
}
```

A felcserélő műveletet önálló, swap nevű függvényként írtuk meg, mivel a qsort három helyen is használja.

```
/* swap: v[i] és v[j] felcserélése */
void swap(int v[ ], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

A standard könyvtár tartalmazza a qsort egy általános változatát, amellyel bármilyen típusú objektumok rendezhetők.

A rekurzióval tényleges tárterületet nem tudunk megtakarítani (csak a forrásprogram lesz rövidebb), mivel az éppen feldolgozott értékek számára egy veremtárat kell fenntartani. A rekurzív program nem is gyorsabb, viszont előnye, hogy a program szövege tömörebb és a rekurzív programot gyakran egyszerűbb megírni, ill. megérteni, mint a nem rekurzív változatot. A rekurzió különösen kényelmes a rekurzívan definiált adatstruktúrák (pl. fák) esetén, és ezzel a kérdéssel a 6.5. pontban még foglalkozunk.

4.12. gyakorlat. A printd függvényben alkalmazott elgondolást felhasználva írjuk meg az itoa függvény rekurzív változatát! A függvény rekurzív hívásokkal alakítson egy egész számot karaktersorozattá.

4.13. gyakorlat. Írjuk meg az s karaktersorozatot helyben megfordító reverse(s) függvény rekurzív változatát!

4.11. A C előfeldolgozó rendszer

A C nyelv a fordítás önálló első meneteként beiktatható előfeldolgozó rendszerrel bizonyos nyelvi kiterjesztéseket tesz lehetővé. A leggyakrabban használt lehetőség, hogy a fordítás során egy másik állomány tartalmát is beépíthetjük a forrásprogramunkba az #include paranccsal és hogy a #define paranccsal lehetőségünk van egy kulcsszót tetszőleges karaktersorozattal helyettesíteni. Ebben a pontban további lehetőségként még a feltételes fordítással és az argumentumot tartalmazó makrókkal fogunk foglalkozni.

4.11.1. Állományok beépítése

Az állománybeépítés egyszerű lehetőséget kínál a `#define` utasítással létrehozott definíciókból, deklarációkból és más elemekből összeállított részek kezelésére. A programban bárhol előforduló

```
#include "állománynév"
```

vagy

```
#include <állománynév>
```

alakú programsor a fordítás során kicserélődik a megadott nevű állomány tartalmával. Ha az állománynév idézőjelek között volt, akkor az adott állomány keresése ott kezdődik, ahol a rendszer a forrásprogramot megtalálta. Ha a keresett állomáy ott nem található vagy a nevét csúcsos zárójelek között adtuk meg, akkor a keresés egy géptől és rendszertől függő szabály szerint állományról állományra folytatódik. Az így beépített állomány maga is tartalmazhat `#include` sorokat.

Gyakran több `#include` sor van a forrásállomány elején, amely az egész program számára közös `#define` utasításokat és a külső változók `extern` deklarációit tartalmazó állományokat vagy a könyvtári függvények prototípus-deklarációihoz való hozzáférést lehetővé tevő header állományokat (mint pl. az `<stdio.h>`) építi be a programba. (Szigorúan véve a headerek nem szükségképpen állományok, a kezelésük módja géptől és rendszertől függ.)

Az `#include` nagy programok deklarációinak összefogására használható előnyösen. Alkalmazásával garantálható, hogy minden forrásállomány azonos definíciókat és változódeklarációkat használ, amivel kizárható néhány nagyon csúnya hiba. Természetesen, ha egy beépített állományt megváltoztatunk, akkor az összes azt felhasználó állományt újra kell fordítani.

4.11.2. Makróhelyettesítés

Egy definíció általánosan

```
#define név helyettesítő szöveg
```

alakú, és hatására a makróhelyettesítés egyik legegyszerűbb formája indul el: a névvel megadott kulcsszó minden előfordulási helyére beíródik a helyettesítő szöveg. A `#define` utasításban szereplő névre ugyanazok a szabályok érvényesek, mint a változók neveire, a helyettesítő szöveg pedig tetszőleges lehet. Általában a helyettesítő szöveg a sor utasítás után fennmaradó része, de hosszú definíciók több sorban is folytathatók, ha az egyes sorok végére a `\` jelet írjuk. A `#define` utasítással definiált név érvényességi tartománya a definíció helyétől az éppen fordított állomány végéig terjed. Egy definícióban felhasználhatunk korábbi definíciókat is. A helyettesítés csak az önálló kulcsszavakra (nevekre) vonatkozik és nem terjed ki az idézőjelek közötti karaktersorozatokra sem. Például hiába egy definiált név az, hogy YES, nem jön létre a helyettesítés a `printf("YES")` utasításban vagy a `YESMAN` szövegben.

A definícióban bármely névhez bármilyen helyettesítő szöveg hozzárendelhető. Például a

```
#define orokos for (;;) /* végtelen ciklus */
```

sor egy új szót, az `orokos`-t (örökös) definiálja a végtelen ciklust előidéző `for` utasításra.

Lehetőség van argumentumot tartalmazó makrók definiálására is, így a helyettesítő szöveg a különböző makróhívásoknál más és más lesz. Példaképp definiáljuk a `max` nevű makrót a következő módon:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Ez a sor hasonlít egy függvényhíváshoz, de nem az, hanem a `max` makrósoron belüli kifejtése, amelyben a formális paraméter (itt `A` vagy `B`) a megfelelő aktuális argumentummal lesz helyettesítve. Így az a programsor, hogy

```
x = max(p + q, r + s);
```

azzal a sorral helyettesítődik, hogy

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

Mindaddig, amíg az argumentumokat következetesen kezeljük, a makró bármilyen adattípus esetén helyes eredményt fog adni, tehát különböző adattípusukhoz nincs szükség különböző `max` makróra (szemben a függvényekkel, ahol minden adattípushoz saját függvénynek kell tartozni).

Ha jól megfigyeljük a `max` makró kifejtését, akkor észrevehetünk benne egy csapdát. A kifejezést kétszer értékeli ki, ami az inkrementáló–dekrementáló operátorok vagy adatbevitel és adatkivitel esetén hibát (mellékhatást) okoz. Például a

```
max(i++, j++) /* Hibás!!! */
```

sorban a kifejtés hatására a nagyobbik argumentum kétszer inkrementálódik. Ügyelnünk kell a zárójelek használatára is, mert megváltozhat a végrehajtási sorrend. Nézzük meg mi történik, amikor a

```
#define square(x) x * x /* Hibás!!! */
```

makrót `square(z + 1)` alakban hívjuk! A kifejtés után a kifejezésben az `x` helyére `z + 1` kerül, így a kifejezés `z + 1*z + 1` lesz, ami nyilvánvalóan hibás.

Mindezek ellenére a makrók használata nagyon hasznos. Ennek jó gyakorlati példája, hogy az `<stdio.h>` headerben a `getchar` és `putchar` gyakran makróként van definiálva, amivel elkerülhető, hogy futás közben minden egyes karakter feldolgozásánál egy járulékos függvényhívás következzen be. Számos függvény a `<ctype.h>` headerben is makróként van definiálva.

A nevek korábbi definíciója megszüntethető az `#undef` paranccsal, így elérhető, hogy az

```
#undef getchar
int getchar(void) { ... }
```

esetben a `getchar` tényleg egy függvény legyen és ne a makró.

Alapesetben a formális paramétereket nem helyettesíti az előfeldolgozó az idézőjelek közötti karaktersorozatban. Ha viszont a helyettesítő szövegben a paraméter nevét egy # jel előzi meg, akkor a makró kifejtésében az aktuális argumentummal helyettesített paramétert tartalmazó idézőjelek közötti karaktersorozat jelenik meg. Az így kapott karaktersorozatok konkatenációval kombinálhatók. Az előbbieket jól példázza a debug funkcióhoz kidolgozott kiíró makró:

```
#define dprint(kif) printf(#kif " = %g\n", kif)
```

Ha ezt a makrót a

```
dprint(x/y);
```

formában hívjuk, akkor a makró a

```
printf("x/y" " = %g\n", x/y);
```

alakban fejtődik ki, és a karaktersorozatok konkatenálódnak, aminek hatására a végső alakja

```
printf("x/y = %g\n", x/y);
```

lesz.

Az aktuális argumentumon belül az " a \" és a \ a \\ karakterekkel helyettesítődik, így az eredmény egy legális karakteres állandó lesz.

A C előfeldolgozó ## operátorának hatására a makrókifejtés alatt konkatenálódnak az aktuális argumentumok. Ha a helyettesítő szövegben a paraméter mellett ## van, akkor a kifejtés során a paraméter helyettesítődik az aktuális argumentummal, eltávolítódik mellőle a ## és a körülötte lévő üres hely (szóközök), majd ezután újra megvizsgálódik a teljes szöveg. A működést a `paste` makrón mutatjuk be, amely konkatenálja a két argumentumát:

```
#define paste(elso, hatso) elso ## hatso
```

A makrót `paste(nev, 1)` formában használva a `nev1` szöveg generálódik.

A ## operátor beágyazott alkalmazásának szabályai elég bonyolultak, a részleteket az A. Függelékben találhatjuk.

4.14. gyakorlat. Definiáljunk egy `swap(t, x, y)` makrót, amely felcseréli a két `t` típusú argumentumát! (A megoldásban segítségünkre lesz a blokkstruktúra.)

4.11.3. Feltételes fordítás

Az előfeldolgozási folyamat közben kiértékelt feltételes utasításokkal lehetőségünk van magának az előfeldolgozásnak feltételektől függő vezérlésére is. Ennek hatására szelektíven iktathatunk be sorokat a programba, a fordítás (előfeldolgozás) során kiértékelt feltételek értékétől függően.

Az `#if` sor hatására az utána álló állandó egész kifejezés (amely nem tartalmaz `sizeof`, `enum` vagy kényszerített típusú `[cast]` állandókat) kiértékelődik, és ha ennek értéke nem nulla, akkor a következő sorok az első `#endif`, `#elif` vagy `#else` utasításig beépülnek a programba. (Az előfeldolgozó `#elif` utasítása hasonló a C `else if` utasításához.) A `defined` (*név*) kifejezés értéke 1 az `#if` utasításban, ha a *név* már definiálva volt, és 0 különben.

Például, ha biztosak szeretnénk lenni abban, hogy a `hdr.h` állomány tartalma csak egyszer, de egyszer legalább beépül a programba, akkor a `hdr.h` állomány beépítési helyének környezetébe az alábbi utasításokat kell elhelyezni:

```
#if !defined(HDR)
#define HDR

/* ide épül be a hdr.h tartalma */

#endif
```

A `hdr.h` első beépülése a programba definiálja a `HDR` nevet, ezért a következő beépülési kísérletnél a név már definiált, így az előfeldolgozó átugorja az `#endif`-ig terjedő részt. Hasonló módon lehet megakadályozni más állományok többszöri beépítését is. Ha ezt a szerkezetet következetesen használjuk, akkor az egyes header állományok saját maguk beépíthetik a számukra szükséges további header állományokat anélkül, hogy a felhasználónak bármit is tudnia kellene a headerek kapcsolatáról. Az alábbi vizsgálatsorozatban a `SYSTEM` név dönti el, hogy melyik headerváltozatot kell a programba beépíteni:

```
#if SYSTEM == SYSV
   #define HDR "sysv.h"
#elif SYSTEM == BSD
   #define HDR "bsd.h"
#elif SYSTEM == MSDOS
   #define HDR "msdos.h"
#else
   #define HDR "default.h"
#endif
#include HDR
```

Az `#ifdef` és `#ifndef` sorok speciális vizsgálatot végeznek: azt ellenőrzik, hogy az adott név definiált-e vagy sem. Ezek felhasználásával az első példát úgy is írhattuk volna, hogy

```
#ifndef HDR
#define HDR

/* ide épül be a hdr.h tartalma */

#endif
```

Mutatók és tömbök

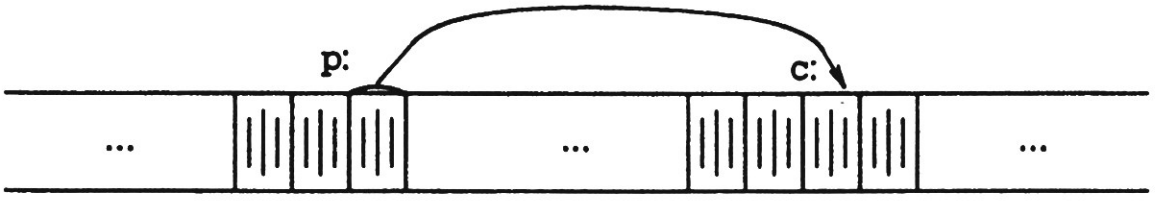
A mutató vagy pointer olyan változó, amely egy másik változó címét tartalmazza. A C nyelvű programokban gyakran használják a mutatókat, egyrészt mert bizonyos feladatokat csak velük lehet megoldani, másrészt mert alkalmazásukkal sokkal tömörebb és hatékonyabb program hozható létre. A mutatók és a tömbök szoros kapcsolatban vannak egymással és ebben a fejezetben ezt a kapcsolatot vizsgáljuk, ill. megmutatjuk, hogy ez a kapcsolat hogyan használható ki.

Gyakran a mutatót összekapcsolják a `goto` utasítással, mondván, hogy mindkettő csodálatos lehetőséget teremt az érthetetlen programok írásához. Ez biztosan így is van, ha nem kellő gondossággal használjuk, hiszen könnyű olyan mutatót létrehozni, amely valamilyen nem várt helyre mutat. Kellő fegyelemmel viszont elérhető, hogy a mutatókat használó program világos és áttekinthető legyen. A következőkben ezt próbáljuk meg bemutatni.

Az ANSI C egyik legfontosabb új eleme, hogy explicit szabályokat tartalmaz a mutatók használatára, amelyeket a jó programozók a gyakorlatban kihasználnak és a jó fordítóprogramok érvényre juttatnak. A korábbi C változathoz képest változás még, hogy a `char *` általános mutató helyett bevezették a `void *` mutatótípust.

5.1. Mutatók és címek

A vizsgálatainkat kezdjük a számítógép tárolójának szervezését leíró egyszerű képpel! A tipikus számítógép tárolója egymást követő, folyamatosan számozott vagy címezett tárhelyekből áll, amelyekkel egyenként vagy folytonos csoportokban végezhetünk műveleteket. Elég gyakori, hogy a tároló bármelyik bájtja egy karakter, bájt-párja egy `short` típusú egész szám és négy szomszédos bájtja egy `long` típusú egész szám tárolására alkalmas. Egy mutató a tárolóelemek egy csoportja (gyakran két vagy négy bájt), amely egy címet tartalmazhat. Ezért, ha `c` egy `char` típusú változó és `p` egy rá mutató (azt címző) mutató, akkor a helyzet a következő vázlatnak megfelelően alakul.



Az `&` unáris (egyoperandusú) operátor megadja egy operandus címét, ezért a

```
p = &c;
```

utasítás `c` címét hozzárendeli a `p` változóhoz és ilyenkor azt mondjuk, hogy `p` `c`-re „mutat". Az `&` operátor csak változókra és tömbelemekre alkalmazható és nem használhatjuk kifejezésekre, állandókra, vagy regisztervátozókra.

A * unáris operátor neve *indirekció*, és ha egy mutatóra alkalmazzuk, akkor a mutató által megcímzett változóhoz férhetünk hozzá. Tegyük fel, hogy x és y egésztípusúak és ip egy int típushoz tartozó mutató. A következő, elég mesterkélt példán bemutatjuk a mutatók deklarálását, valamint az & és * operátorok használatát.

```
int x = 1, y = 2, z[10];
int *ip;      /* ip az int típushoz tartozó mutató */

ip = &x;      /* ip x-re mutat */
y = *ip;      /* y most 1 lesz */
*ip = 0;      /* most x nulla lesz */
ip = &z[0];   /* ip most z[0]-ra mutat */
```

x, y és z deklarációja az eddig látott módon történik, az ip mutatót

```
int *ip;
```

módon deklaráljuk, és azt mondjuk *ip egy int. A változók deklarációjának szintaxisa annak a kifejezésnek a szintaxisát követi, amelyben a változót használjuk. Ezt a meggondolást már eddig is alkalmaztuk a függvények deklarációjánál. Ennek mintájára pl.

```
double *dp, atof(char *);
```

azt jelenti, hogy egy kifejezésben *dp (a dp mutatóval kijelölt változó értéke) és atof értéke double típusú, ill. az atof argumentuma char típushoz tartozó mutató.

Az indirekció alapján látható, hogy egy mutató mindig meghatározott objektumra mutat, azaz minden mutató meghatározott adattípust jelöl ki. (Ez alól csak egy kivétel van, a void típushoz tartozó mutató, ami egy olyan adat, amely bármilyen mutatót tartalmazhat. Erre az a megszorítás érvényes, hogy önmagára nem alkalmazhatja az indirekciót. A kérdésre az 5.11. pontban még visszatérünk.)

Ha ip egy x egészre mutat, akkor *ip minden olyan programkörnyezetben előfordulhat, ahol x használata megengedett. Így pl. megengedett a

```
*ip = *ip + 10;
```

értékadás is, amely *ip-et tízzel növeli.

Az & és * unáris operátorok szorosabban kötnek, mint az aritmetikai operátorok, ezért az

```
y = *ip + 1
```

kifejezés kiértékelésekor a gép először veszi azt az adatot, amire ip mutat, hozzáad egyet, majd az eredményt hozzárendeli y-hoz. Az

```
*ip += 1
```

inkrementálja azt a változót, amire ip mutat, csakúgy, mint a

```
++*ip
```

vagy az

```
(*ip)++
```

Az utolsó esetben a zárójelre szükség van, mert hiányában az `ip` inkrementálódna az `ip` által kijelölt adat helyett, mivel a *-hoz és ++-hoz hasonló unáris operátorok jobbról balra hajtódnak végre.

Mivel a mutatók is változók, ezért indirekció nélkül is használhatók kifejezésekben. Ha pl. `iq` egy `int` adatot címző mutató, akkor

```
iq = ip
```

értékadás hatására `ip` tartalma `iq`-ba másolódik, függetlenül attól, hogy `ip` mire mutat.

5.2. Mutatók és függvényargumentumok

Mivel a C nyelv a függvényeknek érték szerint adja át az argumentumokat, így a hívott függvény nem tudja megváltoztatni a hívó függvény változóit. Például a rendező programban használtuk a `swap` függvényt a rossz sorrendben lévő adatok felcserélésére. Itt nem elegendő, ha azt írjuk, hogy

```
swap (a, b);
```

ahol a `swap` függvényt úgy definiálnánk, hogy

```
void swap(int x, int y) /* Hibás!!! */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Mivel a függvényt érték szerint hívjuk, a `swap` nem képes a hívásában szereplő `a` és `b` argumentumokat befolyásolni (azoknak csak egy *helyi másolatával* dolgozik, ezek cseréje pedig nem befolyásolná az eredeti argumentumok sorrendjét).

A kívánt hatás csak úgy érhető el, ha a hívó függvény *mutatókat* ad át a hívott függvénynek.

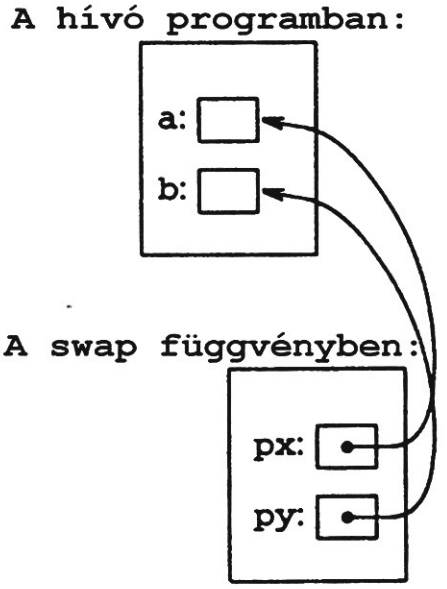
```
swap(&a, &b);
```

Mivel az `&` operátor a változó címét állítja elő, ezért az `&a` az `a`-hoz tartozó mutató. Ha a `swap` függvényben a paramétereket mutatóként deklaráljuk, akkor indirekt módon hozzáfér az operandusokhoz és így közvetlenül felcserélheti azokat. Az így megírt `swap` függvény:

```
void swap(int *px, int *py) /* *px és *py cseréje */
{
    int temp;

    temp = *px;
```

```
        *px = *py;
        *py = temp;
}
```



A folyamatot egyszerű ábrával is szemléltethetjük. Egy függvény a mutató típusú argumentumokon keresztül képes hozzáférni és megváltoztatni a hívó eljárás objektumait. Példaként írjuk meg a `getint` függvényt, amely egy szabad formátumú bemeneti konvertáló eljárás. A `getint` a bemeneti karakteráramból hívásonként egy egész típusú értéket emel ki és visszaadja azt a hívó függvénynek, ill. jelzi a bemeneti állomány végét. A kapott egész számot és az `EOF` jelzését két külön csatornán kell megoldani, mivel az `EOF` jelzése maga is egy egész szám lehet, ami a beolvasott értékkel keveredve zavart okozhatna.

Azt a megoldást választottuk, hogy az `EOF`-ra utaló állapotjelzést a függvény visszatérési értékével adjuk át a hívó programnak, a beolvasott egész számot pedig mutató típusú argumentumon keresztül. (A 7.4. pontban ismertetésre kerülő `scanf` függvény ugyanilyen módon működik.) Az

```
int n, tomb[MERET], getint(int *);

for (n = 0; n < MERET && getint(&tomb[n])
                != EOF; n++)
    ;
```

ciklus `getint` hívásokkal feltölti a tömböt. A `getint` minden hívásakor a beolvasott egész számot elhelyezi a `tomb[n]` pozícióra, majd a hívó ciklus inkrementálja `n` értékét. A helyes működéshez a híváskor a `tomb[n]` tömbelem címét kell a `getint` függvénynek átadni, mert az csak ennek felhasználásával tudja a talált számot visszaadni a hívó eljárásnak.

A `getint` függvény itt közölt változatának visszatérési függvényértéke `EOF`, ha elérte az állomány végét, nulla, ha a következő bemeneti adat nem szám és pozitív érték, ha az eredmény érvényes egész szám (ami a `tomb[n]` helyen van). A program:

```
#include <ctype.h>

int getch(void);
```

```
void ungetch(int);
/* getint: a bemenetről beolvas egy egész számot és a *pn
helyre teszi */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch())) /* átlépi az üres */
        ;       /* helyeket */
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c); /* ez nem szám */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

A *pn-t a teljes `getint` függvényben úgy használjuk, mint egy közönséges `int` típusú változót. A `getint` szintén használja a 4.3. pontban leírt `getch` és `ungetch` függvényeket, mivel a feleslegesen beolvasott karaktert most is vissza kell írni a bemenetre.

5.1. gyakorlat. Ahogy ezt a példaprogramban láttuk, a `getint` az olyan + vagy - előjelet, ami után nem következik számjegy, érvényes, nulla értékű adatként kezeli. Szüntessük meg ezt a problémát úgy, hogy egy nullát visszaírunk a bemenetre!

5.2. gyakorlat. Írjuk meg a `getfloat` függvényt, ami a `getint` lebegőpontos megfelelője! Milyen típusú függvényértékekkel tér vissza a `getfloat`?

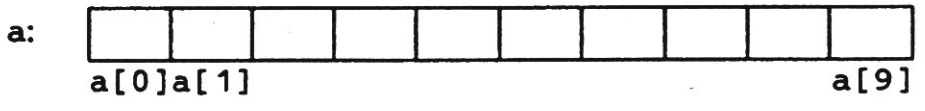
5.3. Mutatók és tömbök

A C nyelvben a mutatók és a tömbök között szoros kapcsolat van, ami indokolja, hogy a két dolgot közösen tárgyaljuk. Bármilyen művelet, amit egy tömb indexelésével elvégezhetünk, megoldható mutatókkal is. A mutatót használó programváltozat általában gyorsabb, de legalábbis a kezdők számára nehezebben érthető.

Az

```
int a[10];
```

deklaráció egy tízelemű tömböt jelöl ki, azaz tíz egymást követő, `a[0], ..., a[9]` névvel ellátott objektumot.



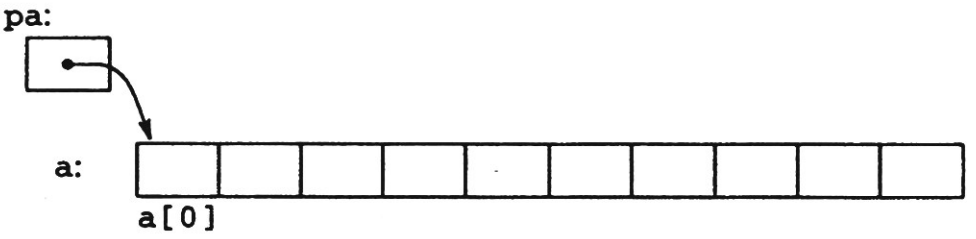
Az a[i] jelölés a tömb i-edik elemére hivatkozik. Ha pa egy egész típushoz tartozó mutató, amit

```
int *pa;
```

módon deklaráltunk, akkor a

```
pa = &a[0];
```

értékadás hatására pa az a tömb nulladik elemére fog mutatni, vagyis pa az a[0] címét fogja tartalmazni.



Most nézzük az

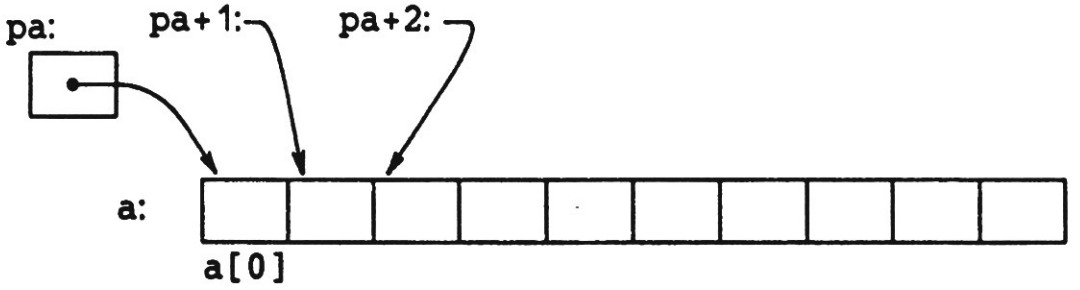
```
x = *pa;
```

értékadást! Ez az a[0] tartalmát fogja az x-be másolni.

Ha pa egy tömb adott elemére mutat, akkor definíció szerint pa+1 a következő elemre, pa+i a pa utáni i-edik elemre és pa-i a pa előtti i-edik elemre fog mutatni. Így ha pa az a[0] elemre mutat, akkor

```
*(pa + 1)
```

a tömb a[1] elemének tartalmára hivatkozik, és pa+i az a[i] címét adja, így a *(pa+i) az a[i] tartalmát jelenti.



Ezek a megállapítások a tömböt alkotó változók típusától vagy méretétől függetlenül igazak. Az a kijelentés, hogy „adj 1-et a mutatóhoz" azt jelenti, hogy pa+1 a következő objektumra, hasonlóan pa+i pedig a pa utáni i-edik objektumra mutat. A teljes mutatóaritmetikára igaz, hogy a növekmény mértékegysége annak az objektumnak a térbeli mérete, amire a mutató mutat.

Az indexelés és a mutatóaritmetikai műveletek közötti összefüggés nagyon szoros. Definíció szerint a tömbre való hivatkozás a tömb első (nulladik indexű) elemét kijelölő mutató létrehozását jelenti. Vagyis a

```
pa = &a[0];
```

értékadás hatására pa és a (mint a tömbre való hivatkozás) azonos értékű lesz. Mivel a tömb neve és a nulladik indexű elemének címe szinonimák, ezért a pa=&a[0] értékadás úgy is írható, hogy

```
pa = a;
```

A fenti megállapítás első látásra nagyon meglepő és azt jelenti, hogy az a[i]-re való hivatkozás *(a+i) formában is írható. Az a[i] hivatkozás kiértékelésekor a C fordító azonnal átalakítja a hivatkozást *(a+i) alakra, és a két alak egymással teljesen egyenértékű. A fenti egyenlőség mindkét oldalára alkalmazva az & operátort az következik, hogy &a[i] és a+i szintén azonosak, vagyis a+i az a utáni i-edik elem címe. Más oldalról nézve viszont ha pa egy mutató, akkor az a kifejezésekben indexelhető, vagyis pa[i] megfelel a *(pa+i)-nek. Röviden összefoglalva: bármely tömböt és indexet tartalmazó kifejezés egyenértékű egy mutatót és egy eltolást (ofszetet) tartalmazó kifejezéssel. A kétféle írásmód egyetlen utasításon belül is megengedett.

A tömb neve és a mutató között csak egyetlen különbség van, amiről nem szabad elfelejtkeznünk: a mutató egy változó, tehát pa = a vagy pa++ érvényes kifejezések, a tömb neve viszont nem változó, ezért az a = pa vagy a++ alakú konstrukciók nem megengedettek!

Amikor egy tömb nevét átadjuk egy függvénynek, akkor valójában a tömb kezdetének címét adjuk át. A hívott függvényben ez az argumentum egy helyi változó lesz, így egy paraméterként megadott tömbnév lényegében egy mutató, vagyis egy címet tartalmazó változó. Ezt a tényt kihasználva írjuk meg egy tetszőleges karaktersorozat hosszát meghatározó strlen függvény egy másik változatát.

```
/* strlen: megadja a karaktersorozat hosszát */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

Mivel s egy mutató, az inkrementálása megengedett, de az s++ nincs semmilyen hatással sem az strlen függvényt hívó függvényben a karaktersorozatra, mivel csak a mutató helyi másolata inkrementálódik. Ez azt jelenti, hogy az strlen függvényt a

```
strlen("Halló mindenki!");     /* karakteres állandó */
strlen(tomb);                  /* char tomb[100]; */
strlen(ptr);                   /* char *ptr; */
```

formában és argumentumokkal híva jól működik.

Egy függvény definíciójában a formális paraméterek között szereplő

```
char s[];
```

és

```
char *s;
```

megadási formák egyenértékűek, de a továbbiakban a másodikat részesítjük előnyben, mert sokkal egyértelműbben mutatja, hogy a paraméter egy mutató. Amikor egy tömb nevét adjuk át a függvénynek, a függvény szabadon dönthet, hogy tömbként vagy mutatóként kezeli (akár mindkét értelmezést is használhatja, ha az célszerű és világos).

Arra is van lehetőség, hogy a tömbnek csak egy részét adjuk át a függvény hívásakor, ha a résztömb kezdetének mutatóját adjuk át. Például, ha `a` egy tömb, akkor

```
f(&a[2])
```

vagy

```
f(a+2)
```

átadja az `f` függvénynek azt a résztömböt, amely az `a[2]` elemmel kezdődik. Az `f` függvényen belül a paraméterdeklaráció

```
f(int rtomb[]) { ... }
```

vagy

```
f(int *rtomb) { ... }
```

alakú lehet. Ami az `f` függvényt illeti, arra semmiféle következménnyel nem jár, hogy a paraméter egy nagyobb tömb része.

Ha biztosak vagyunk benne, hogy a megfelelő tömbelem létezik, akkor megengedett a tömb visszafelé indexelése is. A `p[-1], p[-2]` stb. szintaktikailag helyes és a tömb `p[0]` elemét közvetlenül megelőző elemekre való hivatkozást jelent. Természetesen nem hivatkozhatunk olyan objektumra, ami nincs a tömb határain belül.

5.4. A címaritmetika

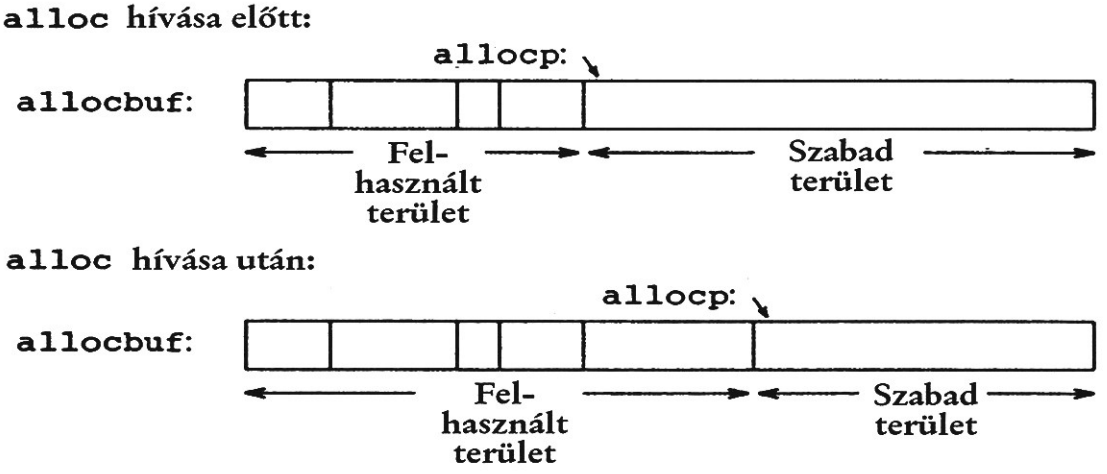
Ha `p` egy tömb valamelyik elemének mutatója, akkor `p++` inkrementálja a `p` mutatót, hogy az a tömb következő elemére mutasson és `p+=i` pedig úgy növeli `p`-t, hogy az az aktuális elem utáni `i`-edik elemre mutasson. Ezek, ill. az ehhez hasonló konstrukciók a mutató- vagy címaritmetika legegyszerűbb esetei.

A C nyelv következetesen és szisztematikusan közelít a címaritmetikához: a mutatók, tömbök és a címaritmetika egységes kezelése a nyelv egyik pozitívuma. Ennek szemléltetésére írjunk egy primitív tárolóhely-kiosztó eljárást. Az eljárás két függvényből fog állni. Az első, `alloc(n)` függvény az `n` darab egymást követő karakterpozícióhoz tartozó mutatóval tér vissza, és ezt az `alloc` függvény hívója a karakterek eltárolásához fogja felhasználni. A második, `afree(p)` függvény felszabadítja a tárterületet, ami így később újra felhasználható lesz. Az eljárást azért neveztük primitívnek, mert az `afree` hívásai az `alloc` hívásaival ellentétes sorrendben kell hogy történjenek. Így az `alloc` és az `afree` által kezelt tárterület lényegében egy veremtár, vagyis egy „utolsó be, első ki" típusú lista. A standard könyvtár hasonló feladatot ellátó `malloc` és `free` függvényeire nincs ilyen megszorítás (ezekről részletesebben majd a 8.7. pontban olvashatunk).

A legegyszerűbb megoldás, ha az `alloc` egy nagy karakteres tömb, az `allocbuf` részeit szolgáltatja. Ez a tömb az `alloc` és `afree` függvényekre nézve saját és közös. Mivel a függvények a feladatot mutatókkal és nem indexekkel oldják meg, ezért egyetlen más eljárásnak sem kell ismerni a tömb nevét, amit az `alloc` és `afree`

függvényeket tartalmazó forrásállományban static tárolási osztályúnak deklaráltunk (ezért más függvényekből nem látható). A gyakorlati megvalósításban nem is fontos, hogy a tömbnek neve legyen, megoldható a feladat úgy is, hogy a `malloc` függvénnyel vagy más módon az operációs rendszertől kérünk egy név nélküli tárterület elejét kijelölő mutatót.

Az `allocbuf` használatához további információk kellenek. A programban egy `allocp` mutatót fogunk használni, ami kijelöli az `allocbuf` következő szabad helyét. Ha az `alloc`-tól `n` karakternyi helyet kérünk, akkor az ellenőrzi, hogy van-e még ennyi szabad hely az `allocbuf`-ban. Ha igen, akkor az `alloc` visszatér az `allocp` aktuális értékével (azaz a szabad terület kezdetével), majd ezután `n` értékével megnöveli, hogy a következő szabad helyre mutasson. Ha az `allocbuf`-ban nincs elegendő hely, akkor az `alloc` nulla értékkel tér vissza. Az `afree` függvény egyszerűen `p` értékre állítja az `allocp` mutatót, ha `p` az `allocbuff`-ba mutat. A puffer kezelését a következő egyszerű ábra mutatja.



A program:

```
#define ALLOCSIZE 10000 /* a rendelkezésre álló hely */

static char allocbuf[ALLOCSIZE]; /* az alloc tárolója */
static *allocp = allocbuf; /* a következő szabad hely */

char *alloc(int n) /* visszatér az n karakterhez tartozó mutatóval */
{
   if (allocbuf + ALLOCSIZE - allocp >= n) {
   /* van elég hely */
      allocp += n;
      return allocp - n; /* a régi mutató */
   } else /* nincs elég hely */
      return 0;
}

void afree(char *p) /* a p-ig terjedő részt felszabadítja */
{
   if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
         allocp = p;
}
```

A mutatót ugyanúgy lehet inicializálni, mint bármelyik más változót, de normális esetben csak a nullát vagy korábban definiált megfelelő típusú adatok címeit tartalmazó kifejezést rendelhetünk hozzá kezdeti értékül. A

```
static char *allocp = allocbuf;
```

deklaráció definiálja az `allocp` karakteres adathoz tartozó mutatót és egyben inicializálja is, hogy az az `allocbuf` kezdetére mutasson, ami a program indulásakor az első szabad hely. Helyette azt is írhattuk volna, hogy

```
static char *allocp = &allocbuf[0];
```

mivel *a tömb neve* egyben *a nulladik indexű elemének a címe.* A program vizsgáló része

```
if (allocbuf + ALLOCSIZE - allocp >= n) {
   /* van elég hely */
```

ellenőrzi, hogy a tömbben van-e elegendő hely `n` karakter számára. Ha igen, akkor az `allocp` legfeljebb eggyel mutat túl az `allocbuf` végén. Ha a helyigény kielégíthető, akkor az `alloc` az `n` karakteres blokk kezdetét kijelölő mutatóval tér vissza (figyeljük meg a függvény deklarációját). Ha nem, akkor az `alloc` egy jelzést ad vissza. A C nyelv garantálja, hogy egy adat címe soha nem nulla, így a visszatéréskor érzékelt nulla függvényérték a normálistól eltérő működést, azaz a szükséges hely hiányát jelzi.

A mutatók és az egész számok nem felcserélhetők. Ez alól egyetlen kivétel van, a nulla: a nulla, mint állandó, hozzárendelhető egy mutatóhoz és egy mutató összehasonlítható a nulla állandóval. A nulla számkonstans helyett gyakran használják a `NULL` szimbolikus állandót, amivel a mutató speciális értékét jelzik. Ezek után a mutatókkal kapcsolatban mi is a `NULL` szimbolikus állandót fogjuk használni. A `NULL` az `<stdio.h>` headerben van definiálva.

Az

```
if (allocbuf + ALLOCSIZE - allocp >= n) {
```

és

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

formájú vizsgálatok a címaritmetika számos fontos tulajdonságára mutatnak rá. Ha `p` és `q` ugyanazon tömb elemeire mutatnak, akkor az `==`, `!=`, `<`, `>`, `<=` stb. relációk helyesen működnek. Például a

```
p < q
```

reláció igaz, ha `p` a tömb egy korábbi (kisebb indexű) elemére mutat, mint `q`. Bármely mutató értelmes módon egyenlőségre vagy nem egyenlőségre összehasonlítható nullával. Ha viszont nem ugyanazon tömb mutatóit használjuk aritmetikai kifejezésekben vagy relációkban, akkor az eredmény értelmetlen (amit vagy azonnal észreveszünk, vagy nem). A szabály alól csak egyetlen kivétel van: egy tömb vége utáni első elem címét a címaritmetika még képes feldolgozni.

Mint már láttuk, egy mutatót és egy egész számot szabad összeadni vagy kivonni. A

```
p + n
```

konstrukció a p mutatóval aktuálisan kijelölt utáni n-edik objektumot jelenti. Ez attól függetlenül igaz, hogy p milyen típusú objektumot címez meg, mivel a fordítóprogram n-t olyan egységekben számolja, mint a p-vel megcímzett objektum mérete (és ezt a p deklarációja határozza meg). Ha pl. az adott számítógépen az int típusú adat négy bájtos, akkor int adatok esetén a mérték négy.

A mutatók kivonása szintén megengedett: ha p és q ugyanazon tömb elemeit címzik, és p < q, akkor q - p + 1 a p és q közötti elemek száma (a határokat is beleértve). Ezt a tényt kihasználva írjuk meg a karaktersorozat hosszát megadó strlen függvény egy újabb változatát!

```
/* strlen: az s karaktersorozat hossza */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

A deklarációban a p kezdeti értékeként s-t adtuk meg, így p a karaktersorozat első elemére mutat. A while ciklusban egyenként vizsgáljuk a karaktereket, amíg el nem érünk a '\0' végértékig. Mivel p karakterekre mutat, p++ mindig a következő karakterre lépteti p-t, és a vizsgálat végén p - s a megvizsgált karakterek számát adja. (A karaktersorozatban lévő karakterek száma túl nagy lehet ahhoz, hogy int típusú adatként kezeljük. Az <stddef.h> headerben definiálva van egy ptrdiff_t típus, ami elegendően nagy ahhoz, hogy két mutató előjeles különbségét tárolja. Ha nagyon gondosak akarunk lenni, akkor az strlen visszatérési típusát a standard könyvtári változatnak megfelelően size_t típusnak választjuk. A size_t egy előjel nélküli egész adattípus, amelyet a sizeof operátor ad visszatérési értékként.)

A címaritmetika működése következetes: ha float típusú adatokkal dolgozunk, amelyek több helyet igényelnek, mint a char típusúak, és p egy float típusú adatot címző mutató, akkor p++ a következő float típusú adatot jelöli ki. Így minden további nélkül megírhatjuk az alloc függvény egy másik változatát, amelyben a char adattípus helyett float adattípust tárolunk. Ehhez mindössze az alloc és afree függvényekben a char szót float-ra kell átírni. Minden mutatókkal végzett művelet automatikusan figyelembe veszi a megcímzett objektum méretét.

A mutatók esetén megengedett műveletek az azonos típusú mutatók közötti értékadás, mutatók és egészek közötti összeadás vagy kivonás, két azonos tömbre értelmezett mutató kivonása vagy összehasonlítása, valamint mutató értékadása vagy összehasonlítása nullával. Minden más mutatókra vonatkozó aritmetikai művelet tilos. Nem lehet két mutatót összeadni, szorozni, osztani, léptetni vagy maszkolni, és ugyancsak tilos mutatóhoz float vagy double típusú értéket hozzáadni. A szabályok alól csak a void * a kivétel, amely rákényszerített típusmegadás (cast) nélkül egy adott típusú mutatóhoz egy másik típusú mutatót rendel.

5.5. Karaktermutatók és függvények

Az

```
"Ez egy karaktersorozat"
```

alakban írt *karaktersorozat állandók* valójában karakterekből álló tömbök, amelyeket a belső ábrázolásban egy null-karakter (`'\0'`) zár. A program a karaktersorozat végét a null-karakter keresésével találja meg. Ezt a belső ábrázolást használva a karaktersorozat tárolásához szükséges hely csak egy karakterrel több, mint az idézőjelek közötti karakterek száma. A karaktersorozat állandók a leggyakrabban függvények argumentumaként fordulnak elő, mint pl. a

```
printf("Halló mindenki!\n");
```

függvényben. Az ehhez hasonló karaktersorozatokhoz a program karakteres mutatón keresztül fér hozzá, azaz a `printf` függvény megkapja a karaktersorozat kezdetének mutatóját. Általánosan igaz, hogy a program a karaktersorozathoz az első elemét kijelölő mutatón keresztül fér hozzá.

A karaktersorozat természetesen nem csak függvények argumentumaként fordulhat elő. Ha a `puzenet` változót úgy deklaráltuk, hogy

```
char *puzenet;
```

akkor a

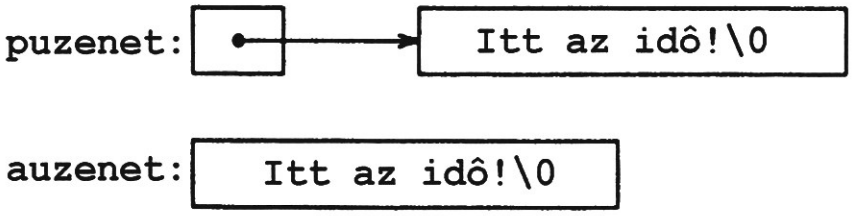
```
puzenet = "Itt az idő!";
```

utasítás a `puzenet`-hez a karaktersorozatot tartalmazó tömb mutatóját rendeli. Ez valójában *nem* karaktersorozat-másolás, csak a mutatók rendelődnek egymáshoz. A C nyelvben nincs olyan operátor, amellyel egy karaktersorozat, mint egyetlen objektum, feldolgozható lenne.

A következő két definíció között lényeges különbség van:

```
char auzenet[] = "Itt az idő!";  /* ez egy tömb */
char *puzenet = "Itt az idő!";       /* ez egy mutató */
```

Az `auzenet` egy tömb, ami elegendően nagy ahhoz, hogy a karaktersorozatot és a `'\0'` végjelzést tárolni tudja, és a tömbnek az `"Itt az idő!"` karaktersorozatot adjuk kezdeti értékül. Másrészt a `puzenet` egy mutató, amelyhez egy karaktersorozat állandót megcímző kezdőértéket rendelünk. Ez a mutató természetesen módosítható, és akkor más objektumot fog megcímezni, viszont a karaktersorozat módosítása definiálatlan eredményt ad. A kétféle definíció térbeli elhelyezkedését a következő vázlat mutatja:



A mutatók és tömbök használatának néhány kérdését két hasznos függvényen keresztül mutatjuk be. A függvények a standard könyvtárban megtalálhatók. Az első, `strcpy(s, t)` függvény a `t` karaktersorozatot az `s` karaktersorozatba másolja át.

Jó lenne, ha azt írhatnánk, hogy `s = t`, de ez csak a mutatót másolja át, magát a karaktersorozatot nem. A karaktersorozat átmásolásához egy ciklus szükséges. A karaktersorozatot átmásoló `strcpy` függvény karakteres tömbökkel megvalósított változata:

```
/* strcpy: a t karaktersorozatot s-be másolja - tömb
indexelésével megvalósított változat */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

A következőkben bemutatjuk az `strcpy` függvény mutatókkal megvalósított változatát:

```
/* strcpy: a t karaktersorozatot s-be másolja - 1. mutatókkal meg-
valósított változat */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Mivel a C nyelv az argumentumokat érték szerint adja át, így az `strcpy` tetszés szerint használhatja az `s` és `t` paramétereket (ill. azok helyi másolatait). A program a szokásos módon inicializálja a mutatókat, majd ezek karakterenként végighaladnak a tömbökön, mindaddig, amíg a `t` karaktersorozatot lezáró `'\0'` át nem másolódik `s`-be.

A gyakorlatban az `strcpy` függvényt nem az előző módon írnánk meg. Egy gyakorlott C programozó inkább a következő változatot részesítené előnyben:

```
/* strcpy: a t karaktersorozat s-be másolja - 2. mutatókkal meg-
valósított változat */
void strcpy(char *s, char *t)
{
    while((*s++ = *t++) != '\0')
        ;
}
```

Ez a változat az `s` és `t` mutatók inkrementálását a ciklus vizsgáló részébe építi be. A `*t++` értéke az a karakter, amelyre `t` az inkrementálás előtt mutat. A postfix inkrementálás a karakter feldolgozásának befejeztéig nem változtatja meg `t`-t. Ugyanígy a karakter a régi `s` által kijelölt pozícióba tárolódik, még `s` inkrementálása előtt. Az átmásolt karaktereket a program a `'\0'` végjelzéssel hasonlítja össze és ez vezérli a ciklust. Mindezek hatására `t` összes karaktere, a lezáró `'\0'` végjelet is beleértve átkerül `s`-be.

A programot vizsgálva megfigyelhető, hogy a `'\0'` végjelzéssel való összehasonlítás redundáns, ezért a függvény még tovább rövidített változata úgy írható be, hogy

```
/* strcpy: a t karaktersorozatot s-be másolja - 3. mutatókkal
megvalósított változat */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Ez a változat első ránézésre nagyon titokzatosnak tűnik, de a jelölés nagyon kényelmes, ezért célszerű elsajátítani. Különböző programokban gyakran találkozhatunk vele.

Az `strcpy` függvény a standard könyvtárban (a `<string.h>` headerben) található, és visszatérési értéke az átmásolt karaktersorozat.

A példaként megírt második függvény az `strcmp(s, t)`, amely az `s` és `t` karaktersorozatokat hasonlítja össze, és visszatérési értéke negatív, nulla vagy pozitív attól függően, hogy az `s` lexikografikusan kisebb, egyenlő vagy nagyobb `t`-nél. (A lexikografikus sorrendet úgy kapjuk, hogy a karaktersorozatokat a gép belső karakterkészletének megfelelően – tágabb értelemben, a különböző jeleket is beleértve – ábécésorrendbe rendezzük. A kisebb, egyenlő vagy nagyobb reláció ekkor az ábécében elfoglalt helyek viszonyát jelzi.) Ezt az értéket úgy kapjuk meg, hogy az első olyan helyen, ahol `s` és `t` különbözik, kivonjuk egymásból a két karaktert. A program tömbindexeléssel megvalósított változata:

```
/* strcmp: visszatérési érték <0, ha s<t, =0, ha s=t
és >0, ha s>t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

A függvény mutatókkal megvalósított változata:

```
/* strcmp: visszatérési érték <0, ha s<t, =0, ha s=t
és >0, s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
    if (*s == '\0')
        return 0;
    return *s - *t;
}
```

Mivel a `++` és `--` prefix és postfix formában egyaránt használhatók, ezért a `*`, ill. a `++` és `--` operátorok, ritkán, de más kombinációban is előfordulhatnak. Például a

```
*--p
```

a p-vel megcímzett karakter elővétele előtt dekrementálja p-t. Még néhány fontosabb kombináció:

```
*p++ = val; /* val értékét a verembe teszi */
val = *--p;  /* a verem tetején lévő elemet val-ba teszi */
```

Az előző két példát érdemes megjegyezni, mivel a verem kezelésének alapműveletei.

Az `strcpy` és `strcmp` függvények deklarációit a standard könyvtár tartalmazza, és több más karaktersorozat-kezelő függvény deklarációjával együtt a `<string.h>` headerben találhatók.

5.3. gyakorlat. Írja meg a 2. fejezetben bemutatott `strcat(s,t)` függvény mutatóval megvalósított változatát! Az `strcat(s,t)` függvény a `t` karaktersorozatot az `s` karaktersorozat végéhez másolja.

5.4. gyakorlat. Írjon `strend(s, t)` néven függvényt, amely 1 értékkel tér vissza, ha a `t` karaktersorozat megtalálható az `s` karaktersorozat végén, és 0 értékkel, ha nem!

5.5. gyakorlat. Írja meg az `strncpy`, `strncat` és `strncmp` könyvtári függvények saját változatát! Ezek a függvények az argumentumként megadott karaktersorozat legfeljebb első `n` karakterével végeznek műveletet, pl. az `strncpy(s, t, n)` a `t` karaktersorozat legfeljebb első `n` karakterét másolja `s`-be. (A könyvtári függvények leírása a B. Függelékben található.)

5.6. gyakorlat. Írjuk át a korábbi fejezetek erre alkalmas példaprogramjait úgy, hogy indexelt tömbök helyett mutatókat használjunk! Erre kiválóan alkalmas az 1. és 4. fejezetben megírt `getline`, a 2., 3. és 4. fejezetben megírt `atoi`, `itoa` és minden változata, a 3. fejezetben használt `reverse`, valamint a 4. fejezetben használt `strindex` és `getop` függvény.

5.6. Mutatótömbök és mutatókat megcímző mutatók

Mivel a mutatók maguk is változók, ezért minden további nélkül tárolhatók tömbökben, csakúgy, mint bármely más változó. Ennek bemutatása céljából írjunk programot, amely szövegsorokat ábécésorrendbe rendez. (Ez a UNIX sort rendezőprogramjának egyszerűsített változata lesz.)

A 3. fejezetben már bemutattuk a Shell-féle algoritmus alapján működő rendezőprogramot és a 4. fejezetben annak javított változatát, a quicksort programot. A példában ugyanezeket az algoritmusokat fogjuk használni, azzal az eltéréssel, hogy most változó hosszúságú szövegsorokat rendezünk az egész számok helyett. Ez lényeges különbség, mivel a szövegsorokat nem hasonlíthatjuk ösze vagy mozgathatjuk egyetlen művelettel. Olyan adatábrázolásra van szükség, ami lehetővé teszi a változó hosszúságú szövegsorok hatékony és kényelmes kezelését.

Ez a mutatókból álló tömbökkel valósítható meg legegyszerűbben. Ha a rendezendő sorokat egy hosszú karakteres tömbben, egymáshoz illesztve tároljuk, akkor az egyes sorok az első karakterüket megcímző mutatón keresztül érhetők el. Ezek a mutatók egy tömbben tárolhatók. Két sor úgy hasonlítható össze, hogy átadjuk a mutatóikat az `strcmp` függvénynek. Ha két sor rossz sorrendben van és fel kell cserélni őket, akkor csak a mutatóikat cseréljük a mutatótömbben, és nem pedig magukat a sorokat. Az elvet a következő ábra szemlélteti:

Ezzel a szervezéssel elkerülhetjük azt a kettős problémát, amit a bonyolult tárolókezelés és a szövegsorok tényleges mozgatásából adódó nagy műveletigény jelent.

A rendezési folyamat három lépésből áll:

az összes sor beolvasása
a sorok rendezése
a rendezett sorok kiíratása

A szokásoknak megfelelően a programot a feladat természetes felosztása szerint tagoljuk, és a `main` csak az egyes programrészeket vezérli. Pillanatnyilag ne foglalkozzunk magával a rendezéssel, hanem koncentráljunk az adatszerkezetre, valamint az adatok bevitelére és kiírására.

Az adatbeviteli programrész beolvassa és eltárolja az egyes sorok karaktereit, valamint ezzel egy időben létrehozza a sorok mutatóit tartalmazó tömböt. Ugyancsak ennek a programrésznek a feladata a sorok számlálása, mivel a sorok számára szükség lesz a rendezés és a kinyomtatás során. Mivel az adatbeviteli programrész csak véges sok sort képes beolvasni, ezért egy adott korlátnál több sor beérkezése esetén –1 értékkel tér vissza (illegális számú sor jelzése).

A sorokat kiíró programrész olyan sorrendben fogja kiírni a szövegsorokat, amilyen sorrendben a mutatóik előfordulnak a mutatótömbben. A program eddig tárgyalt részei:

```
#include <stdio.h>
#include <string.h>

#define MAXSOR 5000
/* max. ennyi sor rendezhető */
char *sorptr[MAXSOR];    /* mutatótömb a sorokhoz */

int readlines(char *sorptr[], int nsor);
void writelines(char *sorptr[], int nsor);
void qsort(char *sorptr[], int bal, int jobb);
/* beolvasott sorok rendezése */
main()
{
    int nsor; /* a beolvasott sorok száma */

    if ((nsor = readlines(sorptr, MAXSOR)) >=0) {
            qsort(sorptr, 0, nsor-1);
            writelines(sorptr, nsor);
            return 0;
    }  else {
            printf("Hiba: túl sok rendezendő sor\n");
            return 1;
        }
}
```

```
#define MAXHOSSZ 1000 /* a sor max. hossza */
int getline(char *, int);
char *alloc(int);

/* readlines: sorokat beolvas */
int readlines(char *sorptr[], int maxsor)
{
    int hossz, nsor;
    char *p, sor[MAXHOSSZ];

    nsor = 0;
    while ((hossz = getline(sor, MAXHOSSZ)) > 0)
        if (nsor >= maxsor || (p = alloc(hossz)) == NULL)
            return -1;
        else {
            sor[hossz-1] = '\0'; /* törli az újsor-karaktert */
            strcpy(p, sor);
            sorptr[nsor++] = p;
        }
    return nsor;
}

/* writelines: kiírja a rendezett sorokat */
void writelines(char *sorptr[], int nsor)
{
    int i;

    for (i = 0; i < nsor; i++)
        printf("%s\n", sorptr[i]);
}
```

A program használja az 1.9. pontban leírt `getline` függvényt.

A legfontosabb újdonsággal a `sorptr` deklarációjában találkozunk. A

```
char *sorptr[MAXSOR]
```

azt mondja, hogy a `sorptr` egy tömb, amelynek `MAXSOR` számú eleme van és minden elem egy `char` típushoz tartozó mutató. Így a `sorptr[i]` egy karakterhez tartozó mutató és `*sorptr[i]` pedig az `i`-ediknek eltárolt szövegsor első karakterének mutatója.

Mivel `sorptr` maga is egy tömb neve, mutatóként kezelhető ugyanúgy, mint a korábbi példákban a tömbnevek, ezért a `writelines` függvényt úgy is megírhatjuk, hogy

```
/* writelines: kiírja a rendezett sorokat */
void writelines(char *sorptr[], int nsor)
{
    while (nsor-- > 0)
        printf("%s\n", *sorptr++);
}
```

Kezdetben `*sorptr` az első sorra mutat, majd az inkrementálás hatására a következő sorra lép, amíg csak `nsor` le nem számlálódik.

Miután a sorok beolvasását, ill. kiíratását elintéztük, rátérhetünk a rendezésre. A 4. fejezetben bemutatott quicksort programot kissé meg kell változtatnunk: módosítani kell a deklarációkat és az összehasonlítást az `strcmp` függvény hívásával kell elvégeznünk. Az algoritmus változatlan marad, ezért bízhatunk benne, hogy továbbra is működni fog.

```
/* qsort: a v[bal] ... v[jobb] tömb rendezése növekvő
sorrendbe */
void qsort(char *v[], int bal, int jobb)
{
    int i, utolso;
    void swap(char *v[], int i, int j);

    if (bal >= jobb)      /* semmit nem csinál, ha */
        return;       /* kettőnél kevesebb elemből áll */
    swap(v, bal, jobb, (bal + jobb)/2);
    utolso = bal;
    for (i = bal + 1; i <= jobb; i++)
        if (strcmp(v[i], v[bal]) < 0)
            swap(v, ++utolso, i);
    swap(v, bal, utolso);
    qsort(v, bal, utolso-1);
    qsort(v, utolso+1, jobb);
}
```

A swap függvényt is csak triviális módon kell megváltoztatni, a deklaráció értelemszerű módosításával:

```
/* swap: v[i] és v[j] felcserélése */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Mivel `v` (a `sorptr`) bármelyik egyedi eleme egy karakteres mutató, így `temp` is az kell, hogy legyen az értékadás miatt.

5.7. gyakorlat. Módosítsuk a `readlines` függvényt úgy, hogy a beolvasott sorokat a `main` által létrehozott tömbben tárolja, és ne az `alloc` függvényen keresztül kérjen mindig helyet a sor számára! A program mennyivel lesz gyorsabb, ha elmarad az `alloc` hívása?

5.7. Többdimenziós tömbök

A C nyelv lehetővé teszi a derékszögű többdimenziós tömbök alkalmazását, bár ezeket sokkal ritkábban használják, mint a mutatótömböket. Ebben a pontban bemutatjuk a többdimenziós tömbök tulajdonságait.

Vizsgáljuk meg a hónap napjairól az év napjaira vagy fordítva történő adatátalakítás feladatát! Például március 1. egy nem szökőév 60. napja, szökőévben pedig a 61. nap. Az átalakításhoz definiáljunk két függvényt: a `day_of_year` függvény a hónap napjait az év napjaivá alakítja, a `month_day` függvény pedig az év napjait a hónap napjaivá. Mivel a `month_day` függvény két értéket (hónap és nap) számol, így a hónap és nap argumentum mutató lesz. A

```
month_day(1988, 60, &h, &n)
```

függvényhívás `h` értékét 2-re, `n` értékét 29-re állítja be (1988. február 29.).

Mindkét függvénynek azonos információra van szüksége: egy táblázatra, ami tartalmazza az egyes hónapokban lévő napok számát. Mivel a hónapok napjainak száma más és más szökőévben és nem szökőévben, ezért egyszerűbb a szökőév és nem szökőév adatait egy kétdimenziós tömb két sorában tárolni, mint mindig vizsgálni, hogy februárban milyen adattal kell dolgozni. Az adatátalakítást végző függvények és az átalakításhoz szükséges tömb:

```
static char naptab[2][13] = {
{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
{0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year: a hónap és nap értékéből kiszámítja
az év napját */
int day_of_year(int ev, int ho, int nap)
{
    int i, szoko;

    szoko = ev%4 == 0 && ev%100 != 0 ||
                    ev%400 == 0;
    for (i = 1; i < ho; i++)
        nap += naptab[szoko][i];
    return nap;
}
/* month_day: az éven belüli napból megadja a hónapot
és a napot */
void month_day(int ev, int evnap, int *pho, int *pnap)
{
    int i, szoko;

    szoko = ev%4 == 0 && ev%100 != 0 ||
              ev%400 == 0;
    for (i = 1; evnap > naptab[szoko][i]; i++)
        evnap -= naptab[szoko][i];
    *pho = i;
    *pnap = evnap;
}
```

Emlékezzünk vissza a logikai kifejezések számértékéről mondottakra: az nulla, ha a kifejezés hamis, és egy, ha igaz. Ezt használtuk ki a `szoko` meghatározásánál, és az így kapott 0 vagy 1 érték felhasználható a `naptab` indexelésére.

A `naptab` tömb a `day_of_year` és `month_day` függvények számára külső változó, így mindkettő használhatja. A `naptab` tömb elemeit `char`-ként adtuk meg, hogy ezzel is megmutassuk a `char` típusú adat nemcsak karakterek, hanem kis egész számok tárolására is alkalmas.

A `naptab` az első kétdimenziós tömb, amivel eddig találkoztunk. A C nyelvben a kétdimenziós tömb valójában egy egydimenziós tömb, amelynek mindegyik eleme szintén egy tömb. Ezért kell az indexeket

```
naptab[i][j] /* [sor][oszlop] */
```

alakba írni a más nyelvekben megszokott

```
naptab[i, j] /*HIBÁS!!!*/
```

alak helyett. A kétdimenziós tömbben az elemek sorfolytonosan tárolódnak, ezért a jobbra az utolsó index (oszlopindex) változik a leggyorsabban, ha az elemeket a tárolás sorrendjében címezzük.

A tömböt kapcsos zárójelek között elhelyezett kezdeti értékekkel inicializáljuk, és a kétdimenziós tömb egyes sorait a megfelelő allisták inicializálják. A `naptab` tömböt egy nulla tartalmú oszloppal kezdtük, hogy az 1-től 12-ig terjedő hónapszámmal természetes módon indexelhessük a 0...11 indexek helyett. Mivel a példában a felesleges oszlop nem növeli számottevően a program helyfoglalását, ezért inkább ezt a megoldást választottuk, mint az indexek átkódolását.

Ha egy kétdimenziós tömböt átadunk egy függvénynek, akkor a függvényben lévő paraméterdeklarációban meg kell mondani az oszlopok számát. A sorok száma közömbös, mivel (az egydimenziós tömbökhöz hasonlóan) itt is a sorok alkotta tömb mutatójat adjuk át (ami jelen esetben 13 `int` típusú adatot tartalmazó tömbre mutat). Ezért ha a `naptab` tömböt pl. egy `f` függvénynek adnánk át, akkor az `f` deklarációja

```
f(int naptab[2][13]) { ... }
```

alakú, amit úgy is írhatnánk, hogy

```
f(int naptab[][13]) { ... }
```

mivel a sorok száma közömbös, vagy

```
f(int (*naptab)[13]) { ... }
```

alakban, ami azt fejezi ki, hogy a paraméter egy mutató, ami 13 egész adatból álló tömbre mutat. A kerek zárójel szükséges, mivel a `[]` szögletes zárójelek nagyobb precedenciájúak, mint a `*`. A zárójel nélküli

```
int *naptab[13]
```

deklaráció 13 mutatóból álló tömböt deklarálna, amelynek mindegyik eleme egy egész típusú adatra mutat. Az ehhez hasonló összetett deklarációk kérdésével az 5.12. pontban még foglalkozunk.

5.8. gyakorlat. A `day_of_year` és `month_day` függvényekben nincs hibaellenőrzés. Küszöböljük ki ezt a hiányosságot!

5.8. Mutatótömbök inicializálása

Írjunk egy `month_name(n)` függvényt, amely egy, az n-edik hónap nevét tartalmazó karaktersorozatot címző mutatót ad visszatérési értékként. Ez a belső `static` tárolási osztályú tömbideális alkalmazási lehetősége! A `month_name` függvény karaktersorozatokból álló saját tömböt tartalmaz, és hívásakor a megfelelő mutatóval tér vissza. A feladat megoldása kapcsán megmutatjuk, hogy hogyan inicializálható a nevekből álló tömb.

A szintaxis hasonló a korábban használt inicializálások szintaxisához:

```
/* month_name: visszatér az n-edik hónap nevével */
char *month_name(int n)
{
    static char *nev[] = {
        "Illegális hónapnév",
        "Január", "Február", "Március",
        "Április", "Május", "Június",
        "Július", "Augusztus", "Szeptember",
        "Október", "November", "December"
    };

    return (n < 1 || n > 12) ? nev[0] : nev[n];
}
```

A karakteres mutatók tömbjét alkotó `nev` deklarációja ugyanolyan, mint a `sorptr` deklarációja volt a rendezőprogramban. Az eltérés csak az, hogy a kezdeti értékek most karaktersorozatok, amelyek hozzá vannak rendelve a tömb megfelelő eleméhez. Az `i`-edik karaktersorozat valahol a tárolóban helyezkedik el (nem tudjuk, hogy pontosan hol, de ez nem is érdekes) és a mutatója van a `nev[i]` helyen. Mivel a `nev` tömb méretét nem specifikáltuk, a fordítóprogram megszámolja a kezdeti értékeket és a kívánt helyre beírja a helyes számot.

5.9. Mutatók és többdimenziós tömbök

A kezdő C programozókat gyakran megzavarja a kétdimenziós tömb és a mutatókból álló tömb (mint pl. a `nev` az előző példában) közötti különbség. Ha pl. adott a következő két definíció:

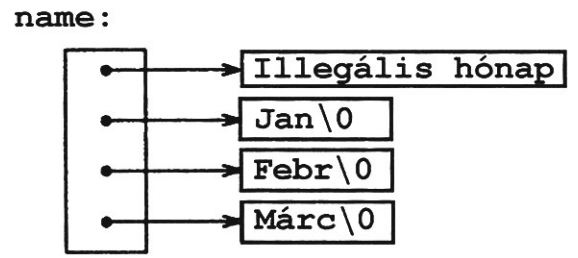
```
int a[10][20];
int *b[10];
```

akkor az `a[3][4]` és `b[3][4]` mindegyike szintaktikailag helyes hivatkozás egy `int` típusú adatra. De `a` valójában egy kétdimenziós tömb: a definíció szerint 200 `int` méretű tárolóhelyet foglalunk le a számára, és egy elemének tényleges helyét mátrixos indexeléssel, mint 20**sor*+*oszlop* határozzuk meg az `a[`*sor*`][`*oszlop*`]` logikai indexelés alap-ján. A `b`-re vonatkozó definíció csak 10 mutató számára foglal helyet és nem rendel

hozzájuk kezdeti értéket. Az inicializálást explicit módon, statikusan vagy a programban kell elvégeznünk. Feltételezve, hogy b minden eleme egy 20 elemű tömbre mutat, akkor ez a tárolóban 200 int változónyi helyet igényel, amihez még hozzájön a 10 mutató helyigénye. A mutatótömbnek van egy lényeges előnye: az általa címzett tömb sorai különböző hosszúságúak lehetnek. Így a b egyes elemei nem szükségképpen mutatnak egy 20 elemű vektorra, lehet olyan, amelyik 2 elemre, a másik 5 elemre, sőt olyan is, amelyik 0 elemre mutat.

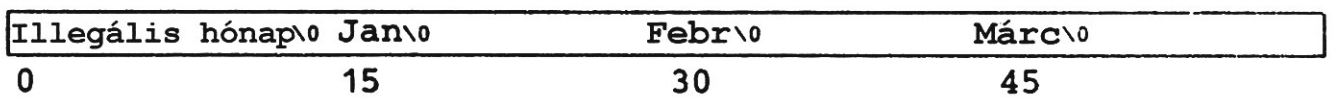
Bár az előbbi fejtegetésben mi mindig egész típusú adatokról beszéltünk, a mutatótömbök leggyakoribb alkalmazása mégis az, amikor elemeik különböző hosszúságú karaktersorozatokra mutatnak (mint pl. a month_name függvényben). Az elmondottak jól láthatók az alábbi deklarációk és a hozzájuk tartozó, tárbeli elhelyezkedést szemléltető ábrák alapján. A mutatótömb:

```
char *nev[] = { "Illegális hónap", "Jan",
"Febr", "Márc" };
```



A kétdimenziós, karaktersorozatokat tároló tömb:

```
char anev[][15] = { "Illegális hónap", "Jan", "Febr",
"Márc" };
```



5.9. gyakorlat. Módosítsuk a day_of_year és month_day függvényeket úgy, hogy indexelés helyett mutatókat használjanak!

5.10. Parancssor-argumentumok

A C nyelvet támogató környezetben lehetőségünk van a programnak parancssor-argumentumokat vagy paramétereket átadni a végrehajtás megkezdésekor. Amikor a végrehajtás kezdetekor a rendszer a main-t hívja, akkor a hívásban két argumentum szerepel. Az első (amit argc-nek szokás nevezni) megadja a parancssor-argumentumok számát, amellyel a programot hívtuk. A második (amit argv-nek szokás nevezni) egy karaktersorozatokat tartalmazó tömböt címző mutató. A tömb tartalmazza a program hívásakor átadandó parancssor-argumentumokat (minden argumentum egy karaktersorozat). Ezeket a karaktersorozatokat általában többszintű mutatóhasználattal kezeljük.

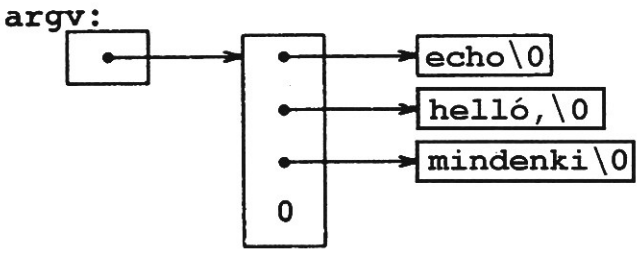
Az elmondottakat a legegyszerűbben az echo program mutatja, amely egyszerűen visszaírja az egy sorban megjelenő, egymástól szóközzel elválasztott parancssor-argumentumokat. Az a parancs, hogy

```
echo Halló mindenki!
```

egyszerűen kiírja a kimenetre, hogy

```
Halló mindenki!
```

Megállapodás szerint az `argv[0]` az a név, amellyel a programot hívták, így `argc` legalább 1. Ha `argc` egy, akkor a program neve után nincs parancssor-argumentum. A mi példánkban `argc` értéke három, és az `argv[0]`, `argv[1]`, ill. `argv[2]` rendre az `"echo"`, `"Halló"`, ill. `"mindenki!"` karaktersorozatokat tartalmazza. A sorban az első opcionális argumentum az `argv[1]` és az utolsó az `argv[argc-1]`. Mindezeken kívül a szabvány megköveteli, hogy `argv[argc]` `NULL` értékű mutató legyen. Az elmondottakat az alábbi ábra szemlélteti.



Az `echo` program első változata az `argv`-t karakteres mutatók tömbjeként kezeli.

```
#include <stdio.h>
/* parancssor-argumentumok visszaírása - 1. változat */
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i],
            (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Mivel `argv` egy mutatótömb mutatója, ezért célszerűbb minden műveletet mutatókkal és nem indexelt tömbökkel végezni. A program következő változata ezért `char` típusú adatokat címző mutatók mutatóját, az `argv`-t inkrementálja, a `argc`-t pedig lefelé számlálja.

```
#include <studio.h>

/* parancssor-argumentumok visszaírása - 2. változat */
main(int argc, char *argv [])
{
    while(--argc, >0)
        printf("%s%s", *++argv,
            (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Mivel argv az argumentumok karaktersorozataiból álló tömb kezdetét kijelölő mutatók mutatója, inkrementálása (++argv) után az eredeti argv[0] helyett az argv[1]-re fog mutatni. argv az egymást követő inkrementálások hatására mindig a következő argumentumot fogja megcímezni és *argv ekkor az argumentum mutatója lesz. Ugyanakkor argc dekrementálódik, és ha értéke nulla lesz, akkor már nincs további kiírandó argumentum.

A printf utasítást úgy is írhattuk volna, hogy

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

Ez szintén azt példázza, hogy a printf argumentuma kifejezés is lehet.

A második példánk a 4.1. pontban ismertetett mintakereső program bővített változata. Ha visszagondolunk a programra, akkor emlékezhetünk rá, hogy a keresett minta mélyen a programba van beágyazva (karakteres állandóként), ami nem túl szerencsés megoldás. A következőkben a UNIX rendszer grep segédprogramjának elvét követve a programot úgy változtattuk meg, hogy a keresendő mintát a parancssor első argumentumaként adhassuk meg. A módosított find program:

```
#include <stdio.h>
#include <string.h>
#define MAXSOR 1000
int getline(char *sor, int max);

/* find: az 1. argumentumában megadott mintát tartalmazó
sorokat megkeresi és kiírja */
main(int argc, char *argv[])
{
    char sor[MAXSOR];
    int talalt = 0;

    if (argc != 2)
        printf("Mintakeresés\n");
    else
        while (getline(sor, MAXSOR) > 0)
            if (strstr(sor, argv[1])
                        != NULL) {
                printf("%s", sor);
                talalt++;
            }
    return talalt;
}
```

A standard könyvtár strstr(s,t) függvénye egy mutatóval tér vissza, amely a t karaktersorozat s-beli első előfordulásának helyére mutat, vagy a NULL értékkel, ha t nem fordul elő s-ben. Az strstr függvény a <string.h> headerben van deklarálva.

Most a programot finomítsuk tovább, hogy újabb mutatókkal kapcsolatos példákat készíthessünk. Tegyük fel, hogy a program hívásakor két argumentumot engedünk meg. Az egyik jelentse azt, hogy „írj ki minden sort, kivéve azokat, amelyekben a keresett minta megtalálható", a másik pedig azt, hogy „írd ki minden sor elé a sorszámot".

A UNIX rendszer alatt futó C nyelvű programok esetén a programnév után opcionálisan megadható paraméterek vagy jelzők (flagek) megállapodás szerint a mínusz jellel kezdődnek. Ha a -x paramétert választjuk az inverz kiírási feltétel jelzésére

(ne írja ki azokat a sorokat, amiben megtalálható a minta) és -n paramétert a sorszámozás jelzésére, akkor a teljes parancs:

```
find -x -n minta
```

alakú lesz, és hatására kiíródik minden olyan sor, amelyben a keresett minta nem található meg és a sorok előtt megjelenik a sorszám.

Az opcionális argumentumok sorrendje tetszőleges kell legyen, és a program további (nem a paramétereket feldolgozó) részének működése nem függhet a megadott argumentumok számától. A felhasználók számára kényelmes lehet, hogy az argumentumok kombinálhatók a

```
find -xn minta
```

alakban. A továbbfejlesztett program:

```
#include <stdio.h>
#include <string.h>
#define MAXSOR 1000

int getline(char *sor, int max);

/* find: kiírja azokat a sorokat, amelyekben az
1. argumentumban megadott minta megtalálható */
main(int argc, char *argv[])
{
    char sor[MAXSOR};
    long sorszam = 0;
    int c, kiveve = 0, szam = 0, talalt = 0;

    while (--argc > 0 && (*++argv)[0])=='-')
        while (c = *++argv [0])
            switch (c) {
            case 'x':
                kiveve = 1;
                break;
            case 'n':
                szam = 1;
                break;
            default:
                printf("find: illegális opció
                    %c\n", c);
                argc = 0;
                talalt = -1;
                break;
            }
    if (argc != 1)
        printf("find -x -n minta\n");
    else
        while (getline(sor, MAXSOR) > 0) {
            sorszam++;
            if ((strstr(sor, *argv) != NULL)
```

```
                    != kiveve) {
                if (szam)
                    printf("%ld:", sorszam);
                printf("%s", sor);
                talalt++;
            }
        }
    return talalt;
}
```

Minden egyes opcionális argumentum elővétele előtt az `argc` dekrementálódik és az `argv` inkrementálódik. A ciklus végén – ha nem volt hiba – az `argc` tartalma megmondja, hogy hány argumentum maradt feldolgozatlanul és `argv` ezek közül az elsőre mutat. Így `argc` akár 1 is lehet, és ekkor `*argv` a keresendő mintára fog mutatni. Mivel `*++argv` a karaktersorozatként megadott argumentum mutatója, ezért a `(*++argv)[0]` a karaktersorozat első karaktere. (Egy másik, szintaktikailag helyes forma az első karakter kijelölésére a `**++argv`.) Mint már említettük, a `[]` szorosabban kötődik, mint a `*` és a `++`, ezért a zárójelekre szükség van. Elhagyva azokat a kifejezés `*++(argv[0])` formában értékelődne ki, ami mást jelent. Más a helyzet, amikor a belső ciklust használjuk, aminek az a feladata, hogy végighaladjon egy kiválasztott argumentum karaktersorozatán! Itt a `*++argv[0]` az `argv[0]` mutatót inkrementálja!

Az itt bemutatottaknál ritkán használunk bonyolultabb mutatós kifejezéseket. Ha mégis szükség lenne ilyenekre, akkor célszerű azokat két vagy három egyszerűbb lépésre bontani.

5.10. gyakorlat. Írjuk meg az `expr` programot, amely kiértékeli a parancssor-argumentumban megadott fordított lengyel jelölésmódú kifejezést! A parancssorban az egyes operátorokat és operandusokat szóköz választja el egymástól, pl. az

```
expr 2 3 4 + *
```

formában, ami a `2*(3+4)` kifejezésnek felel meg.

5.11. gyakorlat. Módosítsuk az 1. fejezetben megírt `entab` és `detab` programot úgy, hogy a tabulátorbeállítási pozíciók listáját a parancssor-argumentumból vegye! Használjuk az alapesetnek megfelelő működést, ha nincs argumentum!

5.12. gyakorlat. Bővítsük az `entab` és `detab` programokat úgy, hogy értelmezni tudják az

```
entab -m +n
```

rövidített jelölést! A bővített forma jelentése, hogy az `m`-edik oszloptól kezdve iktasson be tabulátorokat minden `n`-edik oszlophoz. A program a felhasználó szempontjából kényelmes módon működjön, ha nem adunk meg argumentumot!

5.13. gyakorlat. Írjuk meg a `tail` programot, amely kinyomtatja az utolsó `n` bemeneti sort! Alapfeltételezés szerint legyen `n = 10`, de tegyük lehetővé `n` változtatását egy opcionális argumentummal pl. a

```
tail -n
```

formában. (Ennek hatására az utolsó `n` sor íródjon ki.) A program viselkedjen ésszerűen akkor is, ha a bemenet vagy az `n` értéke ésszerűtlen. A programot úgy írjuk meg, hogy a lehető legjobban használja a rendelkezésére álló tárterületet: a szövegsorokat a rendezőprogramnál leírt módon tároljuk és ne rögzített méretű kétdimenziós tömbként.

5.11. Függvényeket megcímző mutatók

A C nyelvben a függvények ugyan nem változók, de azért lehetséges hozzájuk mutatót definiálni, amivel minden, a mutatókra megengedett művelet elvégezhető (szerepelhet értékadásban, elhelyezhető egy tömbben, átadható egy függvénynek argumentumként, lehet a függvény visszatérési értéke stb.). A függvényekhez rendelt mutatók használatát a korábban megírt rendezőprogram módosított változatán mutatjuk be. A rendezőprogramot alakítsuk át úgy, hogy a `-n` opcionális argumentum hatására a bemeneti sorokat ne lexikografikusan, hanem numerikusan rendezze.

A rendezés általában három részből áll: összehasonlításból, ami meghatározza bármely objektumpár sorrendjét; cseréből, ami megfordítja az objektumok sorrendjét, valamint a rendező algoritmusból, ami mindaddig végzi az összehasonlítást és cserét, amíg minden objektum nem kerül a megfelelő sorrendbe. A rendező algoritmus független az összehasonlítás és a cserélés működésétől, így különböző összehasonlító és cserélő függvényeket használva különböző kritériumok szerint rendezhetünk. Ezt a lehetőséget használjuk ki az új rendezőprogram kialakításánál.

Két szövegsor lexikografikus összehasonlítását eredetileg a `strcmp` végezte. Most szükségünk lesz a `numcmp` függvényre, amely két sort a numerikus értéke alapján hasonlít össze és ugyanolyan módon tér vissza, mint a `strcmp`. Az összehasonlító függvényeket a `main` előtt deklaráljuk, és a megfelelő függvény mutatóját adjuk át a `qsort` függvénynek (ami a rendező algoritmust valósítja meg). Nem foglalkozunk a hibás argumentumok kezelésével, csak a fő feladatra, a mutatók átadására koncentrálunk.

```
#include <stdio.h>
#include <string.h>
#define MAXSOR 5000 /* a rendezhető sorok max. száma */
char *sorptr[MAXSOR]; /* a szövegsorok mutatói */

int readlines(char *sorptr[], int nsor);
void writelines(char *sorptr[], int nsor);

void qsort(void *sorptr[], int bal, int jobb,
                        int (*comp) (void *, void *));
int numcmp(char *, char *);

/* a bevitt sorok rendezése */
main(int argc, char *argv[])
{
    int nsor; /* a beolvasott sorok száma */
    int numeric = 0; /* 1, ha numerikus a rendezés */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nsor = readlines(sorptr, MAXSOR)) >= 0) {
        qsort ((void **) sorptr, 0, nsor-1,
            (int (*)(void*, void*)) (numeric ?
                        numcmp : strcmp));
        writelines(sorptr, nsor);
        return 0;
    } else {
```

```
            printf("Túl sok rendezendő sor\n");
            return 1;
        }
    }
```

A qsort hívásában a strcmp és numcmp függvények címei szerepelnek. Mivel ezek a nevek biztosan függvények nevei, ezért nincs szükség az & operátorra, ugyanúgy, ahogy a tömbök nevei előtt sem.

Úgy módosítottuk a qsort programot, hogy képes legyen bármilyen adattípus feldolgozására, ne csak karaktersorozatokéra. Amint ezt a függvényprototípusban jeleztük, a qsort egy mutatókból álló tömböt, két egész típusú adatot és egy függvényt (két mutató típusú argumentummal) vár. A mutató típusú argumentumok megadásához a void * általános (generikus) mutatótípust használtuk. Bármely mutató átalakítható void * típusúvá, ill. abból visszaalakítható bármiféle információvesztés nélkül, ezért használtunk a qsort hívásánál void * típusú argumentumokat. A megfelelően kialakított típusmegadások garantálják a mutatók típusegyeztetését. Ez a program tényleges megvalósítására semmilyen hatással sincs, de biztosítja a fordítóprogram helyes működését.

Ezután nézzük a qsort függvény módosított változatát!

```
/* qsort: a v[bal] ... v[jobb] rendezése növekvő sor-
rendbe */
void qsort(void *v[], int bal, int jobb,
                  int (*comp)(void *, void *))
{
    int i, utolso;
    void swap(void *v[], int, int);
    if (bal >= jobb) /* nem csinál semmit, ha a tömb */
        return; /* kettőnél kevesebb elemből áll */
    swap(v, bal, (bal + jobb)/2);
    utolso = bal;
    for (i = bal+1; i <= jobb; i++)
        if ((*comp) (v[i], v[bal]) < 0)
            swap(v, ++utolso, i);
    swap(v, bal, utolso);
    qsort(v, bal, utolso-1, comp);
    qsort(v, utolso+1, jobb, comp);
}
```

A deklarációkat különös gonddal kell tanulmányozni!

A qsort negyedik paramétere, az

```
int (*comp)(void *, void *)
```

azt mondja ki, hogy a comp egy függvényt címző mutató, amelynek két void * típusú argumentuma van és int típusú értékkel tér vissza.

A comp függvény használata az

```
if ((*comp)(v[i], v[bal] < 0)
```

sorban összhangban van a deklarációval: comp egy függvényhez tartozó mutató, így *comp maga a függvény, és

```
(*comp)(v[i], v[bal])
```

pedig annak hívása. A zárójelek feltétlenül szükségesek a helyes végrehajtási sorrend céljából. Ha elhagynánk őket, akkor az

```
int *comp(void *, void *) /* HIBÁS!!! */
```

definícióhoz jutnánk, ami azt mondja ki, hogy comp egy függvény, amely egy int típusú adatot megcímző mutatót ad vissza. Ez nyilvánvalóan mást jelent, mint az eredeti értelmezés.

Korábban már bemutattuk a két karaktersorozatot összehasonlító strcmp függvényt, így most csak a numcmp függvénnyel foglalkozunk. A numcmp az első jegytől indulva számérték szerint hasonlít össze két karaktersorozatot. A numcmp az összehasonlításhoz a számokat tartalmazó karaktersorozatot az atof függvénnyel alakítja numerikus változóvá.

```
#include <stdlib.h>

/* numcmp: s1 és s2 karaktersorozat összehasonlítása
numerikusan */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

A két adatot a mutatóik felcserélésével megcserélő swap függvény azonos a fejezet elején leírttal, kivéve, hogy a deklarációk void * típusra változtak.

```
void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Az itt bemutatotton kívül még számtalan más opció is illeszthető a rendezőprogramhoz, ezek közül néhány jó gyakorló feladat lesz.

5.14. gyakorlat. Módosítsuk a rendezőprogramot úgy, hogy kezelni tudja a -r jelzést, amivel a fordított (csökkenő) irányú rendezést írjuk elő! Biztosítsuk, hogy a -r működjön a -n opcióval együtt is!

5.15. gyakorlat. A rendezőprogramot egészítsük ki a `-f` opcióval, ami egyesíti a nagy- és kisbetűket úgy, hogy a rendezésnél nem tesz különbséget közöttük! (Például `A` és `a` összehasonlítva legyen egyenlő.)

5.16. gyakorlat. A rendezőprogramot egészítsük ki a `-d` opcióval, aminek hatására csak a betűk, számjegyek és szóközök kerülnek összehasonlításra! Gondoskodjunk róla, hogy a `-d` opció működjön a `-f` opcióval együtt is!

5.17. gyakorlat. Egészítsük ki a rendezőprogramot mezőkezelési funkcióval, ami lehetővé teszi, hogy a rendezést sorokon belül kijelölt mezőkön hajtsuk végre! Engedjünk meg az egyes mezőkhöz egymástól független opciókészletet. (A könyv eredeti kiadásának tárgymutatóját kulcsszavakra a `-df`, oldalszámokra a `-n` opcióval rendezte egy hasonló rendezőprogram.)

5.12. Bonyolultabb deklarációk

A C nyelvet gyakran bírálják a deklarációinak szintaxisa miatt, különösen a függvényekhez tartozó mutatók használata esetén. A szintaxis megkísérli összeegyeztetni a deklarációt a gyakorlati alkalmazással, ami az egyszerűbb esetekben jól megoldható, viszont a bonyolultabb esetekben zavarokhoz vezethet. Ennek főleg az az oka, hogy egy deklaráció nem olvasható egyszerűen balról jobbra és túl sok a zárójel is. Az

```
int *f();    /* egy int típusú adatot címző mutatóval
                visszatérő függvény */
```

és

```
int (*pf)(); /* pf: egy int típusú értékkel visszatérő
                függvény mutatója */
```

deklarációk közötti különbség jól mutatja a problémát. A * egy prefix operátor, ami alacsonyabb precedenciájú, mint a `()`, ezért zárójeleket kell alkalmazni a megfelelő végrehajtási sorrend érdekében. Bár a valóban bonyolult deklarációk csak ritkán fordulnak elő a gyakorlatban, fontos, hogy értelmezni tudjuk őket és ha szükséges, képesek legyünk létrehozni ilyen deklarációkat. Jó módszer a deklarációk kis lépésekben történő felépítésére a `typedef` parancs, amelyen majd a 6.7. pontban foglalkozunk. Egy másik lehetőség, amivel most fogunk megismerkedni, egy programpár, amelyek egyike az érvényes C nyelvű deklarációt szöveges leírássá alakítja, ill. a másik a szöveges leírásból C nyelvű deklarációt hoz létre. A programmal kapott szöveges leírás már balról jobbra olvasható.

Az első, `dcl` nevű program a bonyolultabb, és ennek feladata a C-beli deklarációk szavakra fordítása úgy, ahogy ez az alábbi példákban* látható:

```
char **argv
   argv: pointer to pointer to char
int (*daytab)[13]
   daytab: pointer to array[13] of int
int *daytab[13]
   daytab: array[13] of pointer to int
```

* Itt a példát meghagytuk az eredeti formájában, mivel a magyarítás (fordítás) teljes átírást igényelt volna a deklarációk szóbeli megfogalmazásának magyar mondatszerkezetre való átalakítása miatt. Magyar mondatszerkezet esetén a kiírt deklarációk sokkal bonyolultabban kombinálhatók össze értelmes mondatokká. Reméljük, az angol változat senkinek nem fog gondot okozni. (A fordító)

```
void *comp
   comp: function returning pointer to void
void (*comp)()
   comp: pointer to function returning void
char(*(*x())[])()
   x: function returning pointer to array[] of
   pointer to function returning char
char(*(*x[3])())[5]
   x: array[3] of pointer to function returning
   pointer to array[5] of char
```

A `dcl` program az A. Függelék 8.5. pontjában részletesen leírt deklarátor által specifikált grammatikán alapszik. A deklarátor (rövidítve dcl) egyszerűsített alakja:

dcl: *opcionális* * *direkt-dcl*

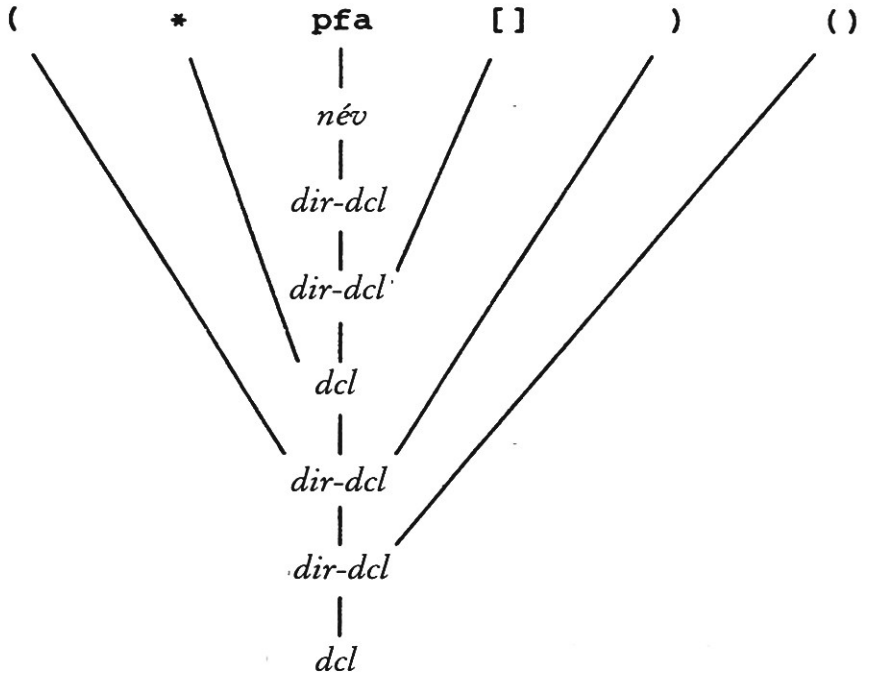
direkt-dcl: *név*
(dcl)
direkt-dcl ()
direkt-dcl [*opcionális méret*]

Szavakba foglalva, a *dcl direkt-dcl,* ha (esetleg) megelőzi egy *. Egy *direkt-dcl* egy név, vagy egy zárójelezett *dcl,* vagy egy *direkt-dcl,* amit zárójel követ, vagy egy *direkt-dcl,* amit szögletes zárójel és opcionális méretmegadás követ.

Ez a grammatika a deklarációk elemzésére használható. Példaképpen nézzük a következő deklarátort:

```
(*pfa[])()
```

Ebben `pfa`-t, mint egy *nevet* azonosíthatjuk és így *direkt-dcl* típusú. Ekkor `pfa[]` szintén *direkt-dcl.* Ezután a `*pfa[]`-ról felismerjük, hogy *dcl* típusú, ezért `(*pfa[])` *direkt-dcl.* Ekkor `(*pfa[])()` egy *direkt-dcl* és így a definíció szerint az egész kifejezés *dcl* típusú. Az elemzés menetét az alábbi elemzőfával szemléltethetjük (a *direkt-dcl* helyett a *dir-dcl* rövidítést használva):



A dcl program legfontosabb része a dirdcl és dcl függvénypár, amelyek a vázlatosan ismertetett grammatika szerint elemzik a deklarációt. Mivel a grammatika rekurzívan van definiálva, ezért az egyes függvények is rekurzívan hívják egymást mindaddig, amíg fel nem ismerik a deklaráció egy darabját. Ennek megfelelően ez egy rekurzívan leszálló elemző program.

```
/* dcl: egy deklarátor elemzése */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*'; )
        ns++; /* a *-ok számolása */
    dirdcl();
    while (ns-- > 0)
        strcat(out, " pointer to");
}
/* dirdcl: egy direkt deklarátor elemzése */
void dirdcl(void)
{
    int type;

        if(tokentype == '(') { /* (dcl) */
        dcl();
        if (tokentype != ')')
            printf("error: missing )\n");
    } else if (tokentype == NAME) /* változó név */
        strcpy(name, token);
    else
        printf("error: expected name or (dcl)\n");
    while ((type=gettoken()) == PARENS ||
                    type == BRACKETS)
        if (type == PARENS)
            strcat(out, " function returning");
        else {
            strcat(out, " array");
            strcat(out, token);
            strcat(out, " of");
        }
}
```

Mivel a programot csak példának szántuk, nem igazán „bombabiztos", a dcl programban van néhány jelentős megszorítás: az csak egyszerű adattípusokat (mint char vagy int) képes kezelni és a rosszul elhelyezett szóközök is megzavarhatják a működését. Mivel a programban nincs hibaállapot-helyreállítás, így az érvénytelen deklarációk is hibás működéshez vezetnek. Ezeknek a hibáknak a kijavítását a gyakorlott programozókra bízzuk.

A program globális változói és a main eljárás:

```
#include <stdio.h>
#include <string.h>
```

```
#include <ctype.h>

#define MAXTOKEN 100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int gettoken(void);
int tokentype; /* az utolsó jel típusa */
char token[MAXTOKEN];  /* az utolsó jel karaktersorozata */
char name[MAXTOKEN]; /* az azonosító neve */
char datatype[MAXTOKEN]; /* adattípus = char, int stb. */
char out[1000]; /* a kimenetet tartalmazó karaktersorozat */

main() /* deklarációk megfogalmazása szavakban */
{
    while (gettoken() != EOF) { /* 1. jel a sorban */
        strcpy(datatype, token); /* ez az adattípus */
        out[0] = '\0';
        dcl(); /* a sor további részeinek elemzése */
        if (tokentype != '\n')
            printf("syntax error\n");
        printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}
```

A gettoken függvény átlépi a szóközöket és tabulátorokat, majd megkeresi a következő szintaktikai elemet (tokent) a bemeneti karaktersorozatban. Egy token lehet egy név, egy kerek zárójelpár, egy szögletes zárójelpár (amiben esetleg egy szám áll) vagy bármilyen egymagában álló karakter.

```
int gettoken(void) /* visszatér a következő jellel */
{
    int c, getch(void);
    void ungetch int ;
    char *p = token;
    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getch()) == ')') {
            strcpy (token, "()");
            return tokentype = PARENS;
        } else {
            ungetch (c);
            return tokentype = '(';
        }
    } else if (c == '[') {
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
```

```
            *p = '\0';
            return tokentype = BRACKETS;
        } else if (isalpha(c)) {
            for (*p++ = c; isalnum(c = getch()); )
                *p++ = c;
            *p = '\0';
            ungetch(c);
            return tokentype = NAME;
        } else
            return tokentype = c;
    }
```

A `getch` és `ungetch` függvényeket a 4. fejezetben már ismertettük.

A feladat megfordítása viszonylag egyszerű, főleg ha nem törődünk a feleslegesen generált zárójelekkel. Az `undcl` program az "`x` is a function returning a pointer to an array of pointers to functions returning `char`" (`x char` típusú adatokkal visszatérő függvények mutatóiból alkotott tömb mutatójával visszatérő függvény) alakú szóbeli leírásból, ami az

```
x () * [] * () char
```

karaktersorozattal fejezhető ki, előállítja a

```
char (*(*x())[])()
```

deklarációt. A rövidített bemeneti szintaxis lehetővé teszi, hogy újra a `gettoken` függvényt használjuk. Az `undcl` függvény ugyanazokat a külső változókat használja, mint a `dcl`.

```
/* undcl: a szóbeli megfogalmazást deklarációvá alakítja */
main ()
{
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, " (%s ", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf (temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("invalid input at %s\n", token);
        printf("%s\n", out);
    }
    return 0;
}
```

A programban használt `sprintf` függvény a `printf`-hez hasonló könyvtári függvény, a `printf`-nek megfelelően formátumozza a kiírandó adatokat, de kiírás helyett az első argumentumában (aminek karaktersorozatnak kell lenni) tárolja. Bővebb leírása a 7.2. pontban, ill. a B. Függelékben található.

5.18. gyakorlat. Egészítse ki a `dcl` programot a bemeneti hibákat megszüntető hiba-helyreállító eljárással!

5.19. gyakorlat. Módosítsa az `undcl` programot úgy, hogy ne írjon ki felesleges zárójeleket a deklarációkban!

5.20. gyakorlat. Bővítse ki a `dcl` programot úgy, hogy kezelni tudja a függvényargumentum típusú és `const`-hoz hasonló minősítőket tartalmazó deklarációkat is!

Struktúrák

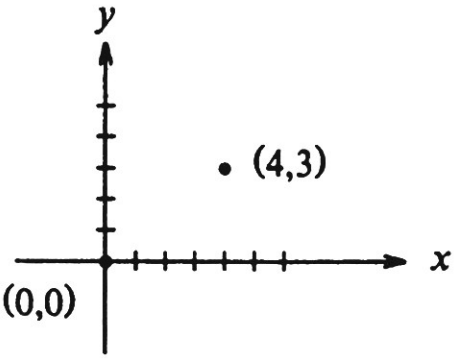
A struktúra egy vagy több, esetleg különböző típusú változó együttese, amelyet a kényelmes kezelhetőség céljából önálló névvel látunk el. Néhány nyelvben az így értelmezett struktúrát rekordnak nevezik (pl. a Pascal rekordja hasonló tulajdonságú adatfajta). A struktúra bevezetése segíti az összetett adathalmazok szervezését, ami különösen nagy programok esetén előnyös, mivel lehetővé teszi, hogy az egymással kapcsolatban lévő változók egy csoportját egyetlen egységként kezeljük, szemben az egyedi adatkezeléssel.

A struktúrára az egyik hagyományos példa a bérszámfejtési lista: ez az alkalmazottakat attribútumok halmazával (név, lakcím, társadalombiztosítási szám, bér stb.) írja le. Ezen attribútumok némelyike maga is lehet struktúra, pl. a név is több részből áll, csakúgy mint a cím vagy a bér. A másik, C nyelvre jellemzőbb példát a számítógépes grafika adja: a pont egy koordinátapárral írható le, a négyzet egy pontpárral adható meg stb.

A struktúrákat érintő, az ANSI szabványból adódó legfontosabb változás, hogy a szabvány értelmezi a struktúrák értékadását. A struktúrák átmásolhatók egymásba, értékül adhatók más struktúráknak, átadhatók függvénynek és a függvények visszatérési értékei is lehetnek. Ezt évek óta a legtöbb fordítóprogram támogatja, de ezeket a tulajdonságokat most pontosan definiáljuk. A szabvány lehetővé teszi az automatikus tárolási osztályú struktúrák és tömbök inicializálását, amivel szintén ebben a fejezetben foglalkozunk.

6.1. Alapfogalmak

Hozzunk létre néhány struktúrát, amelyek a grafikus ábrázoláshoz használhatók. Az alapobjektum a pont, amely egy x és egy y koordinátával adható meg. Tételezzük fel, hogy a koordináták egész számok. A két komponens (koordináta) egy struktúrában helyezhető el a




```
struct pont {
    int x;
    int y;
};
```

deklarációval.

A struktúra deklarációját a `struct` kulcsszó vezeti be, amelyet kapcsos zárójelek között a deklarációk listája követ. A `struct` kulcsszót opcionálisan egy név, az ún. *struktúracímke* követheti (mint a példánkban a `pont`). Ez a címke vagy név azonosítja a struktúrát és a későbbiekben egy rövidítésként használható a kapcsos zárójelek közötti deklarációs lista helyett.

A struktúrában felsorolt változóneveket a struktúra *tagjainak* nevezzük. Egy struktúra címkéje (neve), ill. egy tagjának a neve és egy közönséges (tehát nem struktúra-tag) változó neve lehet azonos, mivel a programkörnyezet alapján egyértelműen megkülönböztethetők. Továbbá ugyanaz a tagnév előfordulhat különböző struktúrákban, bár célszerű azonos neveket csak egymással szoros kapcsolatban lévő adatokhoz használni.

Egy `struct` deklaráció egy típust is definiál. A jobb oldali, záró kapcsos zárójel után következhet a változók listája, hasonlóan az alapadattípusok megadásához. Így pl. a

```
struct { ... } x, y, z;
```

szintaktikailag analóg az

```
int x, y, z;
```

deklarációval, mivel mindkét szerkezet a megadott típusú változóként deklarálja `x`, `y` és `z` változót és helyet foglal számukra a tárolóban.

Az olyan struktúradeklaráció, amelyet nem követ a változók listája, nem foglal helyet a tárolóban, csak a struktúra alakját írja le. Ha a struktúra címkézett volt, akkor a címke a későbbi definíciókban a struktúra konkrét előfordulása helyett használható. Például felhasználva a pont korábbi deklarációját a

```
struct pont pt;
```

definíció egy `pt` változót definiál, ami a `struct pont`-nak megfelelő típusú struktúra. Egy struktúra úgy inicializálható, hogy a definíciót az egyes tagok kezdeti értékének listája követi. A kezdeti értékeknek állandó kifejezéseknek kell lenni. Például:

```
struct pont maxpt = { 320, 200};
```

Egy automatikus struktúra értékadással vagy egy megfelelő típusú struktúrát visszaadó függvény hívásával is inicializálható.

Egy kifejezésben az adott struktúra egy tagjára úgy hivatkozhatunk, hogy

struktúra-név.tag

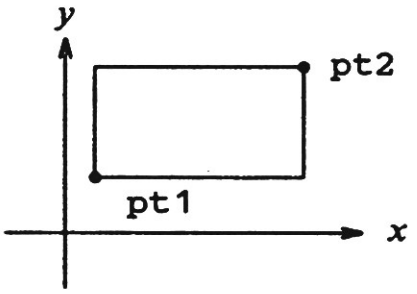
A pont struktúratag operátor összekapcsolja a struktúra és a tag nevét. A `pt` pont koordinátáit pl. úgy írathatjuk ki, hogy

```
printf("%d, %d", pt.x, pt.y);
```

A `pt` pont origótól mért távolsága:

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x +
       (double)pt.y * pt.y);
```

A struktúrák egymásba ágyazhatók. Például egy téglalap az átlója két végén lévő pontpárral írható le,




```
struct tegla {
    struct pont pt1;
    struct pont pt2;
};
```

Ennek alapján a `tegla` struktúra két `pont` struktúrából áll. Ha az `abra` struktúrát úgy deklaráljuk, hogy

```
struct tegla abra;
```

akkor az

```
abra.pt1.x
```

hivatkozás az abra `pt1` tagjának *x* koordinátáját jelenti.

6.2. Struktúrák és függvények

A struktúrák esetén megengedett művelet a struktúra másolása vagy értékadása, ill. a struktúra címéhez való hozzáférés az `&` operátorral és a struktúra tagjaihoz való hozzáférés. Ezek a műveletek a struktúrát egy egységként kezelik, és a másolás vagy értékadás magában foglalja a struktúrák függvényargumentumkénti átadását, ill. a struktúra típusú függvényvisszatérés lehetőségét is. Struktúrák egy egységként nem hasonlíthatók össze. A struktúrák állandó értékek listájával inicializálhatók, és automatikus tárolási osztályú struktúrák kezdeti értéke értékadással is beállítható.

A struktúrák tulajdonságainak vizsgálatához írjunk néhány függvényt, amelyek pontokkal és téglalapokkal manipulálnak. A feladat megoldásának három lehetséges módja van: a függvénynek átadjuk az egyes komponenseket, a teljes struktúrát vagy annak mutatóját. Mindegyik módszernek van előnye és hátránya.

Az első függvény legyen a `makepoint`, amelyet két egész értékkel hívunk és visszatér egy `pont` struktúrával.

```
/* makepoint: egy pont struktúrát csinál az x és y
komponensekből */
struct pont makepoint (int x, int y)
{
    struct pont temp;
    temp.x = x;
    temp.y = y;
```

```
    return temp;
}
```

Vegyük észre, hogy nincs konfliktus abból, hogy az argumentum és a struktúratag neve megegyezik: az összefüggés kiértékelésekor újra felhasználja a rendszer a nevet.

A `makepoint` függvényt bármilyen struktúra dinamikus inicializálására vagy struktúrák függvényargumentumként történő átadására használhatjuk. Például:

```
struct tegla abra;
struct pont kozep;
struct pont makepoint(int, int);

abra.pt1 = makepoint(0, 0);
abra.pt2 = makepoint(XMAX, YMAX);
kozep = makepoint((abra.pt1.x + abra.pt2.x)/2,
                  (abra.pt1.y + abra.pt2.y)/2);
```

A következő lépésben pontokkal aritmetikai műveleteket végző függvényeket hozunk létre. Például:

```
/* addpoint: két pont összeadása */
struct pont addpoint(struct pont p1, struct pont p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

Ennek a függvénynek a két argumentuma és a visszatérési értéke egyaránt struktúra. A függvényben átmeneti változó bevezetése nélkül, közvetlenül a `p1` komponenset növeltük, hogy kihangsúlyozzuk, a struktúra típusú paraméterek éppen úgy érték szerint adódnak át, mint bármely más változó.

A következő példa a `ptinrect` függvény, amely azt ellenőrzi, hogy egy adott pont benne van-e egy téglalapban. Megállapodás szerint a téglalap belsejének tekintjük annak alsó és bal oldali határát, viszont a teteje és a jobb oldali határa már nem része a téglalapnak.

```
/* ptinrect: visszatérés = 1, ha p benne van a tégla-
lapban és = 0, ha nincs */
int ptinrect(struct pont p, struct tegla r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x &&
        p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

A függvény feltételezi, hogy a téglalapot a szokásos formában írtuk le, azaz a `pt1` koordináta kisebb, mint a `pt2`. A következő függvény visszatérési értékül egy ilyen kanonikus alakú téglalapot mint struktúrát ad.

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))
```

```
/* canonrect: kanonizálja a téglalap koordinátáit */
struct tegla canonrect(struct tegla r)
{
    struct tegla temp;

    temp.pt1.x = min (r.pt1.x, r.pt2.x);
    temp.pt1.y = min (r.pt1.y, r.pt2.y);
    temp.pt2.x = max (r.pt1.x, r.pt2.x);
    temp.pt2.y = max (r.pt1.y, r.pt2.y);
    return temp;
}
```

Ha nagy struktúrát kell átadnunk egy függvénynek, akkor sokkal hatékonyabb, ha a struktúra mutatóját adjuk át és nem pedig a teljes struktúrát másoljuk át. Egy struktúrához tartozó mutató éppen olyan, mint egy közönséges változó mutatója. A struktúra mutatójának deklarációja:

```
struct pont *pp;
```

Ez egy `struct pont` típusú struktúrát kijelölő mutatót hoz létre. Ha `pp` egy `pont` struktúrát címez, akkor `*pp` maga a struktúra, és `(*pp).x`, ill. `(*pp).y` pedig a struktúra tagjai. A `pp` értékét felhasználva pl. azt írhatjuk, hogy

```
struct pont kezdet, *pp;

pp = kezdet;
printf("kezdet: (%d, %d)\n", (*pp).x, (*pp).y);
```

A zárójelre a `(*pp).x` kifejezésben szükség van, mert a `.` struktúratag operátor precedenciája nagyobb, mint a `*` operátoré. A `*pp.x` kifejezés azt jelentené, mint a `*(pp.x)`, ami viszont szintaktikailag hibás, mivel jelen esetben `x` nem mutató.

A struktúrák mutatóit gyakran használjuk egy új, rövidített jelölési formában. Ha `p` egy struktúra mutatója, akkor a

`p->`*struktúratag*

kifejezés közvetlenül a struktúra megadott tagját címzi. A `->` operátor a mínusz jel és a nagyobb jel egymás után írásával állítható elő. Ezt felhasználva az előző példa `printf` függvényét úgy is írhatjuk, hogy

```
printf("kezdet: (%d, %d)\n", pp->x, pp->y);
```

A `.` és a `->` operátorok balról jobbra hajtódnak végre, ezért a

```
struct tegla r, *rp = &r;
```

deklaráció esetén az

```
r.pt1.x
rp->pt1.x
(r.pt1).x
```

```
(rp->pt1).x
```

kifejezések egymással egyenértékűek.

A . és -> struktúraoperátorok a függvény argumentumát tartalmazó () kerek és az indexet tartalmazó [] szögletes zárójelekkel együtt a legmagasabb precedenciájú operátorok (l. a 2.1. táblázatot), így rendkívül szorosan kötnek. Például, ha adott a

```
struct {
    int hossz;
    char *str;
} *p;
```

deklaráció, akkor a

```
++p->hossz
```

kifejezés a hossz változót inkrementálja és nem a p-t, mivel a precedenciaszabályoknak és a végrehajtási sorrendnek megfelelő alapértelmezés ++(p->hossz). A kötés zárójelezéssel változtatható meg, pl. a (++p)->hossz a hossz változóhoz való hozzáférés előtt inkrementálja a p értékét, a (p++)->hossz pedig a hozzáférés után inkrementál. Ez utóbbi esetben a zárójelek elhagyhatók.

Ugyanígy a *p->str előkészíti az str által kijelölt adatot, a *p->str++ inkrementálja str-t az általa címzett adat elővétele után és *p++->str pedig inkrementálja p-t, azután, hogy hozzáfért az str által címzett adathoz.

6.3. Struktúratömbök

Írjunk programot, amely megszámolja egy szövegben a C nyelv egyes kulcsszavainak előfordulását! A programban szükségünk lesz egy karaktersorozatokból álló tömbre az egyes kulcsszavak tárolásához, és egy egészekből álló tömbre a számlált értékek tárolásához. Ennek megvalósítására az egyik lehetőség, hogy két független, kulcsszo és kulcsszam nevű tömböt használunk:

```
char *kulcsszo[NSZO];
int kulcsszam[NSZO];
```

Az a tény, hogy a tömbök párhuzamosan léteznek, sugallja egy másik adatszervezési mód, a struktúratömbös megoldás bevezetését. Minden kulcsszóbejegyzés valójában egy adatpárból áll:

```
char *szo;
int szam;
```

és ezekből az adatpárokból létrehozhatunk egy tömböt. Az így deklarált struktúra:

```
struct kulcs {
    char *szo;
    int szam;
} kulcstab[NSZO];
```

Ez a deklaráció létrehoz egy `kulcs` típusú struktúrát és ezzel egy időben definiál egy `kulcstab` tömböt, ami ilyen típusú struktúrákból áll. A definíció egyben le is foglalja a tárterületet a tömb számára. Ennek a tömbnek minden eleme egy struktúra, ezért akár úgy is írhatnánk, hogy:

```
struct kulcs {
    char *szo;
    int szam;
};
struct kulcs kulcstab[NSZO];
```

Mivel a `kulcstab` struktúra a kulcsszavak állandó neveit tartalmazza, a legegyszerűbb, ha külső változóvá tesszük és definiáláskor egyszer és mindenkorra inicializáljuk. A struktúra inicializálása a korábban elmondottak szerint történhet, a definíciót a kezdeti értékek kapcsos zárójelek között elhelyezett listája követi:

```
struct kulcs {
    char *szo;
    int szam;
} kulcstab[] = {
        "auto", 0,
        "break", 0,
        "case", 0,
        "char", 0,
        "const", 0,
        "continue", 0,
        "default", 0,
        /* ... */
        "unsigned", 0,
        "void", 0,
        "volatile", 0,
        "while", 0
};
```

A kezdeti értékeket a struktúratagoknak megfelelő adatpárok formájában soroltuk fel, de sokkal precízebb és szemléletesebb lenne, ha a tömb egyes soraihoz vagy elemi struktúráihoz tartozó kezdeti értékeket fognánk össze a kapcsos zárójelekkel, mint pl.

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
...
```

esetben. Akkor, ha a kezdeti érték egyszerű adat vagy karaktersorozat, ill. az összes kezdeti érték fel van sorolva, akkor a belső kapcsos zárójelek elhagyhatók. Amennyiben a `kulcstab` utáni `[]` között nem szerepel a tömb mérete, akkor a fordítóprogram a kezdeti értékek leszámolásával határozza meg ezt az értéket (ahogyan erre korábban már utaltunk).

A kulcsszavakat számláló program a `kulcstab` definíciójával kezdődik. A `main` eljárás a bemeneti szöveget a `getword` függvény ismételt hívásával olvassa. A `getword` minden hívásakor egy szót olvas és a program minden beolvasott szót a

3. fejezetben leírt bináris keresést végző függvénnyel keres meg a `kulcstab` tömbben. A kereső függvény helyes működéséhez a kulcsszavaknak a tömbben növekvő (azaz ábécé-) sorrendben kell elhelyezkedni.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXSZO 100

int getword (char *, int);
int binsearch (char *, struct kulcs *, int);

/* C kulcsszavait megszámoló program */
main()
{
    int n;
    char szo[MAXSZO];

    while (getword(szo, MAXSZO) != EOF)
        if (isalpha(szo[0]))
            if ((n = binsearch(szo, kulcstab, NSZO)) >= 0)
                kulcstab[n].szam++;
    for (n = 0; n < NSZO; n++)
            if (kulcstab[n].szam > 0)
                printf("%4d %s\n",
                kulcstab[n].szam, kulcstab[n].szo);
    return 0;
}

/* binsearch: a tab[0]...tab[n-1] táblázatban szót
keres */
int binsearch(char *szo, struct kulcs tab[], int n)
{
    int felt;
    int also, felso, kozep;

    also = 0;
    felso = n - 1;
    while (also <= felso) {
        kozep = (also+felso)/2;
        if ((felt = strcmp(szo, tab[kozep].szo)) < 0)
            felso = kozep - 1;
        else if (felt > 0)
            also = kozep + 1;
        else
            return kozep;
    }
    return -1;
}
```

Hamarosan visszatérünk a getword függvényre is. Egyelőre csak annyit érdemes megjegyezni róla, hogy minden alkalommal, amikor beolvas a bemenetről egy szót, azt bemásolja az első argumentumaként megadott tömbbe.

Az NSZO a kulcstab táblázatban lévő kulcsszavak száma. Bár a kulcsszavakat minden további nélkül megszámolhatnánk, sokkal egyszerűbb és biztonságosabb, ha ezt a gépre bízzuk, főképp akkor, ha a lista változhat. Erre az egyik lehetőség, hogy a kezdeti értékek listáját egy null-mutatóval zárjuk és a kulcstab tömböt feldolgozó ciklust addig működtetjük, amíg csak meg nem találtuk a végét.

Ez azonban sokkal több annál, mint amire szükségünk van, mivel a tömb méretét már fordításkor meghatározza a fordítóprogram, és azután már nem változhat. A tömb mérete az egyes bejegyzések méretének és a bejegyzések számának szorzata, és a bejegyzések száma a

```
(kulcstab mérete) / (struct kulcs mérete)
```

művelettel határozható meg. A fordítás idején hatásos sizeof unáris operátor bármilyen C nyelvű objektum méretének meghatározására használható. A

```
sizeof objektum
```

és

```
sizeof (típusnév)
```

kifejezések egy egész számot adnak eredményül, ami a megadott objektum vagy adattípus bájtokban mért mérete. (Szigorúan véve a sizeof operátor egy előjel nélküli egész számot ad, amelynek típus az <stddef.h> headerben definiált size_t típus.) Az objektum változó, tömb vagy struktúra lehet. A típusnév lehet egy alapadattípus neve (pl. int vagy double) vagy egy származtatott típus (pl. struktúra vagy mutató).

A mi esetünkben a kulcsszavak száma a tömb méretének és egy elem méretének hányadosaként adható meg. Ennek kiszámításához és így NSZO megadásához a #define utasítást használhatjuk a következő módon:

```
#define NSZO (sizeof kulcstab / sizeof(struct kulcs))
```

Egy másik lehetőség, hogy a kifejezésbe a tömb méretének és egy meghatározott elem méretének hányadosát írjuk:

```
#define NSZO (sizeof kulcstab / sizeof kulcstab[0])
```

Ez utóbbi alak előnye, hogy akkor sem kell módosítani, ha megváltoztatjuk a tömböt alkotó elemek típusát.

A sizeof operátor nem alkalmazható az #if feltételes fordítási parancsot tartalmazó sorokban, mivel a C előfeldolgozó rendszer nem elemzi a típusneveket, viszont a #define utasítás utáni kifejezésben már szerepelhet, mert ezt nem az előfeldolgozó, hanem a fordítóprogram értékeli ki.

Most még szólnunk kell a getword függvényről! A programunkhoz a pillanatnyilag szükségesnél sokkal általánosabb getword függvényt írtunk, de ettől az még nem lett bonyolultabb. A getword függvény a bemenetről előkészíti a következő „szót", ahol a szót úgy értelmezzük, hogy az betűkből és számjegyekből álló, betűvel kezdődő karaktersorozat vagy egy tetszőleges, nem üres helyet jelentő karakter.

A függvény visszatérési értéke a szó első karaktere, vagy az állomány végét jelző EOF, vagy maga a beolvasott karakter, ha az nem alfabetikus volt.

```
/* getword: beveszi a következő szót vagy karaktert
a bemenetről */
int getword(char *szo, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = szo;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return szo[0];
}
```

A getword függvény használja a 4. fejezetben bemutatott getch és ungetch függvényeket. Amikorra egy alfanumerikus kulcsszó beolvasásra került, akkorra a getword már egy karakterrel többet olvasott be, amit az ungetch hívásával ad vissza a bemenetnek, hogy a getword következő hívásakor rendelkezésre álljon. A getword használja még az üres helyeket átlépő isspace, a betűket azonosító isalpha és a betűket, ill. számjegyeket azonosító isalnum függvényeket is, amelyek a <ctype.h> standard headerben találhatók.

6.1. gyakorlat. Az itt megírt getword függvény nem kezeli az aláhúzást, a karakteres állandót, a megjegyzést (commentet) és az előfeldolgozó rendszert vezérlő sorokat. Írjuk meg a függvény ezen hiányosságokat kiküszöbölő, javított változatát.

6.4. Struktúrákat kijelölő mutatók

A mutatók és a struktúratömbök együttes használatának bemutatására írjuk meg ismét a kulcsszavakat számláló programot úgy, hogy indexelt tömb helyett mutatókat alkalmazunk!

A kulcstab külső deklarációját nem szükséges módosítani, de a main és a binsearch eljárásokat meg kell változtatni. Az új programok:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXSZO 100
```

```
int getword(char *, int);
struct kulcs *binsearch(char *, struct kulcs *, int);

/* C kulcsszavait megszámoló program - mutatós változat */
main()
{
    char szo[MAXSZO];
    struct kulcs *p;

    while (getword(szo, MAXSZO) != EOF)
        if (isalpha(szo[0]))
            if ((p=binsearch(szo, kulcstab, NSZO))
                                        != NULL)
                p->szam++;
    for (p = kulcstab; p < kulcstab + NSZO; p++)
        if (p->szam > 0)
            printf("%4d %s\n", p->szam, p->szo);
    return 0;
}

/* binsearch: a tab[0] ... tab[n-1] táblázatban szót
keres */
struct kulcs *binsearch(char *szo, struct kulcs *tab,
    int n)
{
    int felt;
    struct kulcs *also = &tab[0];
    struct kulcs *felso = &tab[n];
    struct kulcs *kozep;

    while (also < felso) {
        kozep = also + (felso - also)/2;
        if ((felt = strcmp(szo, kozep->szo)) < 0)
            felso = kozep;
        else if (felt > 0)
            also = kozep + 1;
        else
            return kozep;
    }
    return NULL;
}
```

A programokban több dologra is szeretnénk felhívni a figyelmet! Elsőnek a `binsearch` függvény deklarációjára, ami azt jelzi, hogy a függvény egy egész adat helyett egy `struct kulcs` típusú struktúrát megcímző mutatóval tér vissza. Ezt a függvény prototípusában és a `binsearch`-ban egyaránt deklaráltuk. Ha a `binsearch` talál egy szót, akkor egy azt címző mutatóval tér vissza, ha a keresés sikertelen, akkor a visszatérési érték `NULL`.

Másodiknak érdemes megemlíteni, hogy a `kulcstab` elemeit mutatókkal jelöljük ki. Ez a `binsearch` jelentős megváltoztatását igényli. Az `also` és `felso` változókhoz rendelt kezdeti érték mutató lesz, ami a táblázat kezdetére, ill. éppen a vége utáni helyre mutat.

A középső elem helyének kiszámítása sem történhet a megszokott és egyszerű

```
kozep = (also+felso)/2; /* HIBÁS!!! */
```

összefüggéssel, mivel két mutató összeadása tilos. A kivonás művelete megengedett, így `felso-also` az elemek száma (ami `int` típusú) és

```
kozep = also + (felso-also)/2;
```

kifejezés a `kozep` értékét úgy állítja be, hogy az éppen az `also` és `felso` között középen elhelyezkedő elemre mutasson.

A legfontosabb módosítás, hogy az algoritmust úgy kellett átalakítani, hogy garantáltan ne adjon illegális vagy a tömbön kívüli elemet címző mutatót. A probléma az, hogy mind az `&tab[-1]`, mind az `&tab[n]` kívül esik a `tab` tömb határain. Az első szigorúan tilos és a második alak is illegális hivatkozást eredményezhet. Mindazonáltal a C nyelv definíciója garantálja, hogy a címaritmetika egy tömb utolsó utáni elemével (itt `tab[n]`-nel) helyesen működjön.

A `main` eljárásban azt írtuk, hogy

```
for (p = kulcstab; p < kulcstab + NSZO; p++)
```

Ha `p` egy struktúra mutatója, akkor minden `p`-t használó aritmetikai műveletnél a címaritmetika figyelembe veszi a struktúra méretét, így `p++` helyes mértékben inkrementálja a mutatót, hogy az a struktúrákból álló tömb következő elemére mutasson. Ennek következtében a ciklus ellenőrző feltétele a megfelelő időben állítja le a ciklust.

Ne gondoljuk azt, hogy a struktúra mérete az egyes tagok méreteinek összege! Mivel különböző objektumok összeigazításáról van szó, közöttük a struktúrában névvel nem rendelkező lyukak lehetnek! Így pl., ha a `char` típus egy bájtot, az `int` négy bájtot igényel, akkor a

```
struct {
    char c;
    int i;
};
```

struktúra nyolc bájton helyezkedik el, szemben a várt öt bájttal. A `sizeof` operátor mindig a tényleges méretet adja.

Végül még szólnánk a program formájáról: amikor egy függvény egy komplikált típusú adattal, pl. egy struktúra mutatójával tér vissza, mint a

```
struct kulcs *binsearch(char *szo, struct kulcs *tab,
       int n)
```

esetben is, a függvény neve nehezen vehető észre vagy kereshető szövegszerkesztővel. Emiatt néha a

```
struct kulcs *
binsearch(char *szo, struct kulcs *tab, int n)
```

írásmódot szokták alkalmazni. Mindkét forma megfelelő, a választás közöttük ízlés kérdése.

6.5. Önhivatkozó struktúrák

Tételezzük fel, hogy az előbbinél sokkal általánosabb problémát akarunk megoldani: egy bemeneti szöveg *összes* szavának előfordulását akarjuk megszámolni. Mivel a szavak listája eredendően ismeretlen, így a hagyományos rendezés és bináris keresés nem használható. A lineáris keresést, amikor minden beolvasott szóról egy szólistán elölről kezdve végigmenve eldöntenénk, hogy már előfordult-e vagy sem, alkalmazhatnánk, de a program futási ideje rendkívül nagy lenne (pontosabban a futási idő valószínűleg a bemeneti szavak számának négyzetével arányosan növekedne). Hogyan kellene megszerveznünk az adatrendszerünket ahhoz, hogy hatékonyan megbirkózzunk a tetszőleges szavakból álló listával?

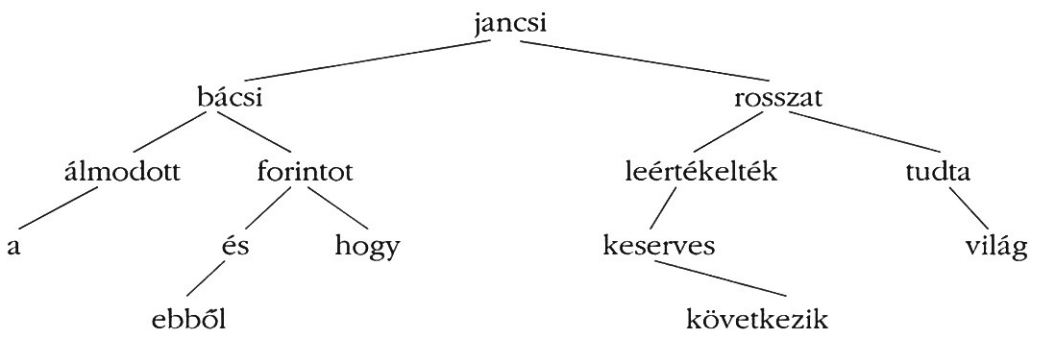
Az egyik lehetséges megoldás, hogy állandóan rendezett állapotban tartjuk a már feldolgozott szavak halmazát úgy, hogy a beérkezés sorrendjében a megfelelő helyre rakjuk a szavakat. Ezt azonban nem úgy csináljuk, hogy egy egyindexes tömbben folyton odébbtoljuk a már meglévő szavakat, hogy helyet szorítsunk egy új szónak. Ez a megoldás szintén nagyon nagy futási időt eredményezne, ezért helyette egy *bináris fának* nevezett adatstruktúrát alkalmazunk.

A bináris fa minden egyes különböző szóhoz egy „csomópontot" rendel, és minden csomópont a

– a szó szövegét megcímző mutatóból,
– a szó előfordulásának számából,
– a csomópont bal oldali gyermekét (leszármazottját) címző mutatóból,
– a csomópont jobb oldali gyermekét (leszármazottját) címző mutatóból

áll. Egy csomópontnak kettőnél több gyermeke nem lehet.

Az egyes csomópontokat úgy generáljuk, hogy bármelyik csomópont bal oldali részfája csupa olyan szót tartalmaz, amely lexikográfiásan kisebb nála, amíg a jobb oldali részfa a lexikográfiásan nagyobb szavakat tartalmazza. Így annak a mondatnak a bináris fája (a kis- és nagybetűk között nem téve különbséget és a vesszőt nem véve figyelembe), hogy „Jancsi bácsi rosszat álmodott, leértékelték a forintot és ebből tudta, hogy keserves világ következik" a következő módon néz ki:



Azt, hogy az éppen beolvasott szó benne van-e a fában, úgy dönthetjük el, hogy elindulunk a gyökértől és a beolvasott szót mindig összehasonlítjuk az adott csomóponton tárolt szóval. Ha bárhol egyezést tapasztalunk, akkor a kérdést igenlő módon megoldottuk. Ha a szó kisebb a tárolt szónál, akkor a keresést a bal oldali gyermekkel, különben pedig a jobb oldalival folytatjuk. Ha a kívánt irányban nincs gyermek, akkor az éppen beolvasott szó nincs a fában és a hiányzó gyermek üres helyére kell beírnunk. Ez a folyamat rekurzívan ismételhető, mivel bármely csomópontból kiinduló keresés a csomópont gyermekéből kiinduló keresést használja. Ezért elég természetes, hogy a szavak beillesztésére, majd az eredmény kiírására rekurzív eljárásokat használunk.

Visszatérve az egyes csomópontok leírásához, látszik, hogy azok kényelmesen reprezentálhatók egy négy tagból álló struktúrával:

```
struct fcsomo {          /* a fa csomópontja */
   char *szo;            /* a szó szövegének mutatója */
   int szam;             /* előfordulások száma */
   struct fcsomo *bal; /* bal oldali gyermek */
   struct fcsomo *jobb; /* jobb oldali gyermek */
};
```

A csomópontoknak ez a rekurzív deklarációja elég megdöbbentően hat, de szintaktikailag korrekt. Az illegális, ha egy struktúra saját magára való hivatkozást tartalmaz, de a

```
struct fcsomo *bal;
```

a `balt`-t, mint az `fcsomo`-hoz tartozó mutatót deklarálja és nem egy `fcsomo` struktúrát.

Alkalmasint meg kell említeni az önhivatkozó struktúrák egy változatát, amikor két struktúra hivatkozik egymásra. Ez a következő módon oldható meg:

```
struct t {
   ...
   struct s *p /* p egy s típusú struktúrát */
};     /* címez */
struct s {
   ...
   struct t *q /* q egy t típusú struktúrát */
};     /* címez */
```

A szavak előfordulását számláló program meglepően kicsi, mivel felhasználja a korábban már megírt `getword` függvényt. A `main` eljárás a `getword` függvénnyel szavakat olvas a bemenetről és az `addtree` függvénnyel elhelyezi azokat a fában.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXSZO 100
struct fcsomo *addtree(struct fcsomo *, char *);
void treeprint(struct fcsomo *);
int getword(char *, int);

/* szavak gyakoriságának számlálása */
main()
{
   struct fcsomo *gyoker;
   char szo[MAXSZO];

   gyoker = NULL;
   while (getword(szo, MAXSZO) != EOF)
      if (isalpha(szo[0]))
         gyoker = addtree(gyoker, szo);
   treeprint(gyoker);
   return 0;
}
```

Az addtree függvény rekurzív. A main egy szót helyez el a fa legfelső szintjén (a gyökéren). Ezután az addtree minden egyes lépésben az új szót összehasonlítja a csomópontban tárolt szóval és az eredménytől függően a bal vagy a jobb oldali ágon halad tovább az addtree rekurzív hívásával. A szó vagy megegyezik egy csomóponton tárolt szóval (ekkor az előfordulások számát tároló szam változót növelni kell eggyel), vagy elérjük a végjelzést (null-mutatót), ami azt jelzi, hogy egy új csomópontot kell létrehozni és a fához csatolni. Ha új csomópontot kellett létrehozni, akkor az addtree az azt címző mutatóval tér vissza, ami bekerül a szülő csomópontjába. Az addtree függvény:

```
struct fcsomo *talloc(void);
char *strdup(char *);

/* addtree: w elhelyezése a p című csomópontban
vagy az alatt */
struct fcsomo *addtree(struct fcsomo *p, char *w)
{
    int felt;

    if (p == NULL) { /* egy új szó érkezett */
        p = talloc(); /* csinál egy új csomópontot */
        p->szo = strdup(w);
        p->szam = 1;
        p->bal = p->jobb = NULL;
    } else if ((felt = strcmp(w, p->szo)) == 0)
        p->szam++; /* megismétlődött a szó */
    else if (felt < 0)
    /* kisebb, a bal oldali részfába kerül */
        p->bal = addtree(p->bal, w);
    else /* nagyobb, a jobb oldali részfába kerül */
        p->jobb = addtree(p->jobb, w);
    return p;
}
```

Az új csomóponthoz szükséges tárolóhelyet a talloc függvény készíti elő és egy mutatóval tér vissza, ami az új csomópont elhelyezésére alkalmas szabad tárolóhelyet címzi. Az új szót az strdup függvény másolja az így kijelölt (de közelebbről nem ismert) helyre. (A talloc és strdup függvényeket később ismertetjük.) Az új csomópontban 1 kezdeti értéket kap az előfordulási darabszám és a két gyermeket kijelölő mutató értéke NULL lesz. Az addtree függvénynek ez a része csak a fán belüli keresés befejezésekor, azaz új csomópont beiktatásakor hajtódik végre. A talloc és a strdup hiba-ellenőrzését elhagytuk, ami persze nem túl szerencsés.

A treeprint függvény szétválogatott formában kinyomtatja a fát. Minden csomópontnál először kiírja a bal oldali részfát (minden, a csomópontbeli szónál kisebb szót), majd a csomópont szavát és ezután a jobb oldali részfát (minden, a csomópontbeli szónál nagyobb szót). Ha az olvasó bizonytalan a rekurzív eljárások használata terén, javasoljuk, hogy a fejezet elején megadott fa felhasználásával papíron, ceruzával szimulálja a treeprint működését.

```
/* treeprint : a p fa rendezett kiírása */
void treeprint(struct fcsomo *p)
```

```
{
    if (p != NULL) {
        treeprint(p->bal);
        printf("%4d %s\n", p->szam, p->szo);
        treeprint(p->jobb);
    }
}
```

Meg szeretnénk jegyezni, hogy ha a fa kiegyensúlyozatlanná válik, mert a szavak nem véletlenszerűen érkeznek, akkor a program futási ideje nagyon megnőhet. A legrosszabb esetben, amikor a szavak már sorrendben vannak, a program az igen időigényes lineáris keresési algoritmust szimulálja. A bináris fáknak vannak olyan általánosításai, amikkel ez a kedvezőtlen viselkedés elkerülhető, de ezekkel itt nem foglalkozunk.

Mielőtt befejeznénk a példa elemzését, érdemes megvizsgálni a tárterület-lefoglalás egyik problémáját. Nyilvánvalóan jó lenne, ha a programban csak egyetlen tárterület-lefoglaló eljárás lenne, amellyel különböző objektumok számára kérhetnénk helyet. Ha azonban ugyanaz az eljárás foglal helyet mondjuk egy `char` típusú adathoz tartozó mutató és egy `struct fcsomo` típusú struktúrához tartozó mutató számára, akkor két kérdés merül fel. Az első, hogy egy valós számítógép esetén hogyan elégítsük ki a különböző objektumokra vonatkozó elhelyezési követelményeket (pl., hogy egy egész típusú adatnak mindig páros tárcímen kell elhelyezkedni). A második kérdés, hogy milyen deklaráció teszi lehetővé, hogy a tárterület-lefoglaló eljárás különböző típusú mutatókkal térhessen vissza.

Az objektumok tárbeli elhelyezésére vonatkozó előírások viszonylag könnyen kielégíthetők, némi helyveszteség árán, ha megköveteljük, hogy a tárterület-lefoglaló eljárás mindig mutatóval térjen vissza, mivel az minden elhelyezési előírást kielégít. Az 5. fejezetben ismertetett `alloc` függvény semmiféle meghatározott elhelyezést nem garantál, ezért a programunkhoz a `malloc` standard könyvtári függvényt használtuk, ami kielégíti ezeket a követelményeket. A 8. fejezetben mutatunk egy lehetőséget a `malloc` megvalósítására.

Egy `malloc`-hoz hasonló függvény típusdeklarációja minden olyan nyelvben gondot okoz, amely a típusellenőrzést komolyan veszi. A C nyelvben a megfelelő módszert az jelenti, ha a `malloc` visszatérési értékét `void` típusú mutatóként deklaráljuk, majd explicit kényszerített típusmódosítással érjük el, hogy a mutató a kívánt típusú legyen. A `malloc` és a vele rokon függvények az `<stdlib.h>` standard headerben vannak deklarálva. A fentiek alapján írt `talloc` függvény a következő:

```
#include <stdlib.h>

/* talloc: létrehoz egy fcsomo csomópontot */
struct fcsomo *talloc(void)
{
    return (struct fcsomo *) malloc(sizeof(struct fcsomo));
}
```

Az `strdup` függvény mindössze az argumentumában megadott karaktersorozatot másolja a `malloc` hívásával kapott helyre.

```
char *strdup(char *s) /* másolatot készít s-ről */
{
    char *p;
```

```
    p = (char *) malloc(strlen(s) + 1);
        /* a +1 hely a '\0' jelnek kell */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

A `malloc` függvény `NULL` értékkel tér vissza, ha nincs szabad hely és a `strdup` továbbadja ezt az értéket, amivel a hívó eljárásra bízza a hibakezelést.

A `malloc` függvény hívásával lefoglalt tárterület a `free` függvény hívásával szabadítható fel és tehető újra felhasználhatóvá. A kérdéssel részletesebben a 7. és 8. fejezetben foglalkozunk.

6.2. gyakorlat. Írjunk programot, ami beolvas egy C nyelvű programot és ábécésorrendben kiírja a változók azon csoportjait, amelyekben az első 6 karakter azonos, de a 7. karaktertől kezdődően valahol különböznek! A program ne vegye figyelembe a karaktersorozatokban és megjegyzésekben lévő szavakat! A feladatot úgy oldjuk meg, hogy a 6 parancssorból megadható paraméter legyen!

6.3. gyakorlat. Írjunk kereszthivatkozási programot, amely kiírja egy dokumentumban lévő szavakat az előfordulás, helyüket megadó sorszámokkal együtt! A program ne vegye figyelembe az olyan töltelékszavakat, mint „a", „az", „és" stb.!

6.4. gyakorlat. Írjunk programot, amely kiírja a beolvasott szöveg egyes szavait azok előfordulási gyakoriságának csökkenő sorrendjében! Minden szó elé írjuk ki az előfordulásának számát is!

6.6. Keresés táblázatban

Ebben a pontban egy táblázatkereső programcsomag gerincét írjuk meg, amivel bemutatjuk a struktúrák néhány további alkalmazási lehetőségét és tulajdonságát. A programcsomag elég tipikus, ilyen programok találhatók a makrófeldolgozók és fordítóprogramok szimbólumtáblázatot kezelő részében. Például nézzük a `#define` utasítást! Amikor valahol előfordul a

```
#define BE 1
```

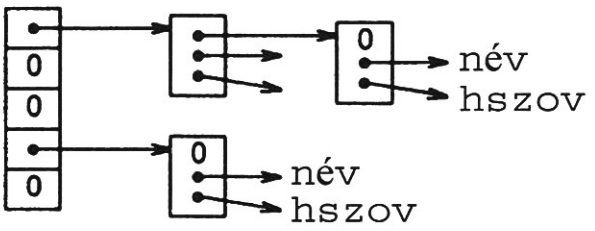
programsor, akkor a `BE` nevet és az `1` helyettesítő szöveget egy táblázatban tárolja el a rendszer. Később, amikor a `BE` név előfordul egy olyan utasításban pl., mint

```
allapot = BE;
```

azt az `1` értékkel kell helyettesíteni.

A nevek és helyettesítő szövegek kezelésére két eljárást írtunk. Az `install(s, t)` függvény beírja az `s` nevet és a `t` helyettesítő szöveget a táblázatba (`s` és `t` mindkettő karaktersorozat). A `lookup(s)` függvény megkeresi `s`-t a táblázatban és egy mutatóval tér vissza, amely `s` táblázatbeli helyére mutat vagy `NULL` értékű, ha `s` nincs a táblázatban.

A kereséshez az ún. *hash algoritmust* használjuk, amely a bejövő nevet kis, nem negatív egész számmá alakítja és ezt a számot használja egy mutatókból álló tömb indexelésére. A tömb egy eleme a hash-kódolású neveket tartalmazó blokkok láncolt listájának kezdetére mutat. A tömbelem értéke `NULL`, ha a hash-kódnak megfelelő értékű név. A lista szerkezetét a következő oldalon található vázlat mutatja.



A lista minden blokkja egy struktúra, amely a név mutatóját, a helyettesítő szöveg mutatóját, valamint a lista következő blokkjának mutatóját tartalmazza. A következő blokk NULL mutatója jelzi a lista végét. A blokkot alkotó struktúra deklarációja:

```
struct nlist { /* a táblázat bejegyzései: */
    struct nlist *kovetkezo; /* a következő elem a listában */
    char *nev; /* a definiált név */
    char *hszov; /* a helyettesítő szöveg */
};
```

A megfelelő mutatótömb:

```
#define HASHMERET 101

static struct nlist *hashtab[HASHMERET];
/* a mutatók táblázata */
```

A lookup és install függvényekben használt hash-kódot előállító függvény a karaktersorozatban lévő karaktereket összegzi az előző összeg megfelelően „összekavart" értékével és az eredmény ennek az összegnek a tömbmérettel való osztásakor kapott maradék. Ez nem a legjobb algoritmus, de mindenesetre rövid és egyszerű. A függvény programja:

```
/* hash: az s karaktersorozat hash-kódját generálja */
unsigned hash(char *s)
{
    unsigned hashert;

    for (hashert = 0; *s != '\0'; s++)
        hashert = *s + 31 * hashert;
    return hashert % HASHMERET;
}
```

Az előjel nélküli típusmegadás (és az ehhez tartozó aritmetikai művelet) garantálják, hogy a hash-kód nem negatív.

A hash-kódolási eljárás létrehoz egy kezdő indexet a hashtab tömbhöz, és ha a karaktersorozat egyáltalán megtalálható valahol, akkor a blokkok itt kezdődő listájában kell lennie. A keresést a lookup függvény végzi. Ha a lookup megtalálja a táblázatban a bejegyzést, akkor annak mutatójával tér vissza, ha nem, akkor a visszatérési érték NULL.

```
/* lookup: s keresése a hashtab táblázatban */
struct nlist *lookup (char *s)
{
    struct nlist *np;
```

```
for (np = hashtab[hash(s)]; np != NULL; np = np->kovetkezo)
    if (strcmp(s, np->nev) == 0)
      return np; /* megtalálta */
   return NULL; /* nem találta meg */
}
```

A `lookup` függvényben lévő `for` ciklus a láncolt listában való haladás szokásos megvalósítása:

```
for (ptr = fej; ptr != NULL; ptr = ptr->kovetkezo)
   ...
```

Az `install` függvény a `lookup`-ot használja annak eldöntésére, hogy a név benne van-e a táblázatban. Ha igen, akkor az új bejegyzés felülírja a régit, ha nem, akkor viszont egy új bejegyzés keletkezik. Az `install` visszatérési értéke `NULL`, ha bármilyen ok miatt nincs hely egy új bejegyzés létrehozására.

```
struct nlist *lookup(char *);
char *strdup (char *);

/* install: a nevet és a helyettesítő szöveget elhe-
lyezi a hashtab-ban */
struct nlist *install(char *nev, char *hszov)
{
   struct nlist *np;
   unsigned hashert;

   if ((np = lookup (nev)) == NULL) { /* nem találta */
      np = (struct nlist *) malloc(sizeof(*np));
      if (np == NULL || (np->nev = strdup(nev))
                              == NULL)
         return NULL;
      hashert = hash (nev);
      np->kovetkezo = hashtab[hashert];
      hashtab [hashert] = np;
   } else /* már van ilyen bejegyzés */
      free ((void *) np->hszov); /* felszabadítja az
            előző helyettesítő szöveg helyét */
   if ((np->hszov = strdup(hszov)) == NULL)
      return NULL;
   return np;
}
```

6.5. gyakorlat. Írjunk `undef` néven függvényt, amely a `lookup` és `install` függvények kezelte táblázatból töröl egy nevet és a hozzá tartozó definíciót!

6.6. gyakorlat. Ebben a részben ismertetett eljárások, valamint a `getch` és `ungetch` függvények felhasználásával valósítsa meg a `#define` utasítást feldolgozó egyszerű (argumentumokat nem használó) programot! A program legyen alkalmas a C nyelv szintaktikája szerinti, argumentum nélküli `#define` utasítások feldolgozására!

6.7. A typedef utasítás

A C nyelv `typedef` utasításával új adattípus-neveket hozhatunk létre. Például a

```
typedef int Hossz;
```

deklaráció bevezeti a `Hossz` típusnevet mint az `int` típusnév szinonimáját. A `Hossz` ezután szabadon használható deklarációkban, kényszerített típuskonverzióban stb., csakúgy, mint az `int` típusnév. Pl.:

```
Hossz len, maxlen;
Hossz *hosszak[];
```

Hasonlóan a

```
typedef char *String;
```

deklaráció hatására a `String` a `char *` karakteres mutató szinonimája lesz, így használható a következő deklarációkban, ill. kényszerített típuskonverzióban:

```
String p, sorptr[MAXSOR], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Vegyük észre, hogy a `typedef`-ben deklarált típus a változó nevének a helyén jelenik meg és nem közvetlenül a `typedef` utasításszó után! Szintaktikailag a `typedef` hasonló a tárolási osztályokat kijelölő `extern, static` stb. utsításokhoz. A `typedef` utasítással deklarált típus nevét nagy kezdőbetűvel írtuk, hogy kiemeljük a szövegkörnyezetből.

Most nézzünk egy bonyolultabb példát! A `typedef` utasítással rendeljünk új típusnevet a korábban használt fa csomópontjaihoz!

```
typedef struct fcsomo *Faptr;

typedef struct fcsomo { /* a fa csomópontja: */
    char *szo; /* a szöveg mutatója */
    int szam; /* az előfordulások száma */
    Faptr bal; /* bal oldali gyermek */
    Faptr jobb; /* jobb oldali gyermek */
} Facsomo;
```

Ez a deklaráció két új típusnevet, a `Facsomo`-t, ami egy struktúra és a `Faptr`-t, ami a struktúra mutatója, vezet be. Ezek felhasználásával a `talloc` függvény új programja:

```
Faptr talloc(void)
{
    return (Faptr) malloc(sizeof(Facsomo));
}
```

Ki kell hangsúlyoznunk, hogy a `typedef` utasítás semmilyen értelemben sem hoz létre új adattípust, mindössze a már meglévő adattípushoz rendel új típusnevet. Szemantikailag sem jelent újat: az így deklarált változók pontosan ugyanolyan tulajdonságúak, mint az

explicit módon deklarált változók. Hatását tekintve a `typedef` hasonló a `#define` utasításhoz, azzal a különbséggel, hogy a `typedef`-et a fordítóprogram dolgozza fel, amely olyan bonyolult szöveges helyettesítésekkel is megbirkózik, amivel a `#define` utasítást feldolgozó előfeldolgozó rendszer nem képes. Például

```
typedef int (*MIF)(char *, char *);
```

deklaráció létrehozza az `MIF` „Mutató `Int` értékkel visszatérő (két `char *` típusú argumentummal hívott) Függvényhez" nevű adattípust, ami olyan összefüggésekben használható, mint

```
MIF strcmp, numcmp;
```

volt az 5. fejezet rendezőprogramjában.

Az esztétikai vonatkozásokon kívül a `typedef` utasítást két okból szokás alkalmazni. Az első, hogy a parametrizálás a program hordozhatóságánál okozhat gondokat. A `typedef`-fel deklarált adattípusok gépfüggetlenné tehetők a `typedef` módosításával. Az általános gyakorlat az, hogy a `typedef`-fel bevezetett neveket különböző egész típusú adatokhoz használjuk, és ekkor az éppen használt számítógép `short`, `int` és `long` adattípusaihoz megfelelő deklarációkat alakítunk ki. Erre jó példát mutatnak a standard könyvtárban deklarált `size_t` és `ptrdiff_t` adattípusok.

A másik ok, amiért a `typedef` utasítást használjuk, hogy az így deklarált típusok javítják a program olvashatóságát, mert pl. a `Faptr` típusnevet könnyebb megérteni, mint egy bonyolult szerkezet mutatójának deklarációját.

6.8. Unionok

Az *union* egy olyan változó, amely különböző időpontokban különböző típusú és méretű objektumokat tartalmazhat úgy, hogy a fordítóprogram ügyel az objektumok méretére és tárbeli elhelyezésére vonatkozó előírások betartására. Az unionok alkalmazásával lehetővé válik, hogy azonos tárterületen különböző fajta adatokkal dolgozzunk, anélkül, hogy a programba géptől függő információkat kellene beépíteni. A C nyelv unionja analóg a Pascal record adattípusának egy változatával.

Az unionok használatának bemutatására szánt példánkat a fordítóprogram szimbólumtábla-kezelőjéből vettük. Tegyük fel, hogy az állandóink `int`, `float` vagy karakteres mutató típusúak lehetnek! Az adott állandó értékét a megfelelő típusú változóban kell tárolnunk, viszont a táblázat kezelése szempontjából az a kényelmes, ha az értékek ugyanannyi tárterületet foglalnak el és ugyanazon a helyen tárolódnak, függetlenül a típusuktól. Ez az union használatának fő célja: egy olyan változó, amely megengedett módon többféle adattípus bármelyikét tárolhatja. Az union szintaxisa a struktúrákon alapszik:

```
union u_tag {
    int iert;
    float fert;
    char *sert;
} u;
```

Az így létrehozott `u` változó elegendően nagy lesz ahhoz, hogy az itt megadott három adattípus közül a legnagyobbat is tárolni tudja. Az `u` specifikált konkrét mérete a használt számítógéptől függ. A megadott három adattípus bármelyike hozzárendelhe-

tő u-hoz és ezután kifejezésekben használható mindaddig, amíg a felhasználás következetes: a művelet végén visszanyert típusnak meg kell egyeznie a legutoljára u-ba tárolt típussal. A programozó feladata, hogy nyomon kövesse, mikor és mit tárolt az unionba. Ha egy adatot adott típusként tárolunk el és más típusként nyerünk vissza, akkor az eredmény géptől függő lesz.

Szintaktikailag az union tagjai az

union-név.tag-név

vagy az

union-mutató->tag-név

módon érhetők el, csakúgy, mint a struktúrák tagjai. Ha az `utype` változót használjuk az `u` unionban éppen tárolt adattípus nyilvántartására, akkor az

```
if (utype == INT)
   printf("%d\n", u.iert);
else if (utype == FLOAT)
   printf("%f\n", u.fert);
else if (utype == STRING)
   printf("%s\n", u.sert);
else
   printf("hibás típus %d az utype-ban\n", utype);
```

programrészlet lehetővé teszi az union tartalmának tárolt típustól függő kiíratását.

Az unionok előfordulhatnak struktúrákban és tömbökben is, és ez megfordítva is lehetséges. Egy struktúrán belüli unionhoz való hozzáférés (vagy az unionon belüli struktúrához való hozzáférés) módja megegyezik a beágyazott struktúrákhoz való hozzáférés módjával. Például az alábbi struktúratömböt definiálva a

```
struct {
   char *nev;
   int jelzok;
   int utype;
   union {
      int iert;
      float fert;
      char *sert;
   } u;
} szimbtab[NSZIMB];
```

az `iert` tagra a

```
szimbtab[i].u.iert
```

formában hivatkozhatunk, és az `sert` karaktersorozat első karakteréhez az alábbi két forma bármelyikével hozzáférhetünk:

```
*szimbtab[i].u.sert
szimbtab[i].u.sert[0]
```

A valóságban az union olyan struktúra, amelyben a tagok közötti, báziscímhez viszonyított eltolás nulla. Ez a struktúra elegendően nagy ahhoz, hogy a legszélesebb tagot is tárolni tudja, miközben teljesül az unionban tárolt összes adattípusra a tárbeli elhelyezkedés minden előírása. Az unionokkal ugyanazok a műveletek végezhetők, mint a struktúrákkal: egy egységként kezelve részt vehetnek értékadásban és egyben másolhatók, előállíthatjuk a címüket és hozzáférhetünk a tagjaihoz.

Egy union csak az elsőként megadott tagjához tartozó típusú értékkel inicializálható, így az előbbi `u` unionhoz csak egész típusú kezdőérték rendelhető.

A 8. fejezetben található tárterület-lefoglaló program bemutatja, hogy egy union használatával hogyan lehet kikényszeríteni egy változó adott típusú tárterülethatárra való illeszkedését.

6.9. Bitmezők

Amikor a tárolóhely a szűk keresztmetszet, gyakran kényszerülünk arra, hogy több objektumot egyetlen gépi szóban helyezzünk el. Erre jó példa a fordítóprogram szimbólumtáblát kezelő része, ahol egybites jelzőket használunk. A kívülről kényszerített adatformátumok (pl. hardvereszközök illesztésekor) is gyakran igénylik egy gépi szó részéhez való hozzáférést.

Képzeljük el a fordítóprogramnak azt a részét, amelyik a szimbólumtáblát kezeli! Minden egyes programbeli azonosítóhoz bizonyos információk tartoznak: kulcsszó vagy sem, külső és/vagy statikus változó vagy sem stb. Ezek az információk legtömörebben egybites jelzőként tárolhatók egyetlen `char` vagy `int` típusú adatban.

Ezt szokásos módon úgy oldjuk meg, hogy definiálunk egy maszkhalmazt a megfelelő bitpozícióhoz, pl. az alábbiak szerint:

```
#define      KULCSZSZO   01
#define      EXTERNAL    02
#define      STATIC      04
```

vagy

```
enum { KULCSZSZO = 01, EXTERNAL = 02, STATIC = 04 };
```

A számértékek bitpozíciót jelölnek ki, így kettő hatványának kell lenniük. Ezek után az egyes bitek a 2. fejezetben leírt bitműveletekkel (léptetés, maszkolás, komplementálás stb.) már hozzáférhetők.

Bizonyos programozási idiómák gyakran előfordulnak, mint pl. a

```
jelzok |= EXTERNAL | STATIC;
```

utasítás, amely 1-be állítja a `jelzok` változóban az `EXTERNAL` és `STATIC` állapotot jelző biteket, a

```
jelzok &= ~(EXTERNAL | STATIC);
```

amely kikapcsolja (0-ba állítja) azokat, vagy az

```
if ((jelzok & (EXTERNAL | STATIC)) == 0) ...
```

amelynek értéke igaz, ha mindkét jelzőbit kikapcsolt állapotú.

Bár ezek az idiómák egyszerűen megjegyezhetők és használhatók, a C nyelv azt is lehetővé teszi, hogy egy gépi szón belül közvetlenül, minden bitenkénti logikai művelet nélkül, bitmezőket definiáljunk és használjunk. Egy bitmező (vagy röviden csak mező) egy számítógéptől függően definiált tárolási egységen (amit szónak fogunk nevezni) belül elhelyezkedő szomszédos bitek halmaza. A bitmező definíciójának és hozzáférésének a szintaxisa a struktúrák használatán alapszik. Például a szimbólumtáblához tartozó korábbi `#define` szerkezetek három bitmező definiálásával helyettesíthetők:

```
struct {
    unsigned int mkulcsszo : 1;
    unsigned int mextern : 1;
    unsigned int mstatic : 1;
} jelzok;
```

Ez a struktúra egy `jelzok` nevű változót definiál, amely három egybites mezőt tartalmaz. A mezőket `unsigned int` típusúnak deklaráltuk, hogy garantáltan előjel nélküli mennyiségek legyenek.

Az egyes bitmezőkre ugyanúgy hivatkozhatunk, mint a struktúrák tagjaira: `jelzok.mkulcsszo`, `jelzok.mextern` stb. A bitmezők a kis egész számokhoz hasonlóan viselkednek, és éppúgy, mint más egész számok, szerepelhetnek aritmetikai kifejezésekben. Így az előbbi példák sokkal természetesebb módon úgy írhatók, hogy:

```
jelzok.mextern = jelzok.mstatic = 1;
```

ami 1-be állítja a biteket,

```
jelzok.mextern = jelzok.mstatic = 0;
```

ami 0-ba állítja a biteket vagy

```
if (jelzok.mextern == 0 && jelzok.mstatic == 0)
    ...
```

ami vizsgálja az állapotukat.

Majdnem minden, amit a bitmezőkkel kapcsolatban elmondtunk, az éppen használt számítógéptől függ. Az, hogy egy mező átlépheti-e a szóhatárt, szintén a géptől függ. A bitmezőknek nem kötelező nevet adni. A név nélküli (csak egy kettősponttal és a szélességgel megadott) bitmezők helykitöltésre használhatók. A speciális, 0 szélességű bitmezővel a következő szóhatárra való illeszkedést kényszerítjük ki.

Néhány számítógépnél a bitmezők kijelölése jobbról balra, más típusoknál balról jobbra történik. Ezért, bár a bitmezők igen hasznosak a belsőleg definiált adatstruktúrák tárolására, a külsőleg definiált (pl. egy hardvereszköztől érkező) adatok szétbontásakor nagyon meg kell gondolni a mezők elhelyezkedését. Sajnos emiatt a programok változhatnak és így nem lesznek hordozhatók. A bitmezők csak mint `int` adattípus deklarálhatók és `signed` vagy `unsigned` specifikátor írható elő rájuk. Ügyeljünk arra, hogy a bitmezők nem tömbök, nem címezhetők és így az `&` operátor sem alkalmazható rájuk!

Adatbevitel és adatkivitel

Az adatbeviteli és adatkiviteli szolgáltatás nem része a C nyelvnek, így idáig nem fordítottunk rá nagy figyelmet. Nyilvánvaló viszont, hogy a programok a környezettel a korábban bemutatottnál sokkal bonyolultabb kapcsolatban is lehetnek. Ezen kapcsolat kialakítása érdekében ebben a fejezetben leírjuk a standard könyvtárat, ami függvények gyűjteménye. Ezek a függvények a C nyelvű programok adatbeviteli és adatkiviteli, karaktersorozat-kezelési, tárkezelési, matematikai műveletekkel kapcsolatos és más egyéb szolgáltatásait látják el.

Az ANSI szabvány ezeket a könyvtári funkciókat pontosan definiálja, így ezek bármely C nyelvet használó számítógép és operációs rendszer számára kompatíbilis formában léteznek. Azok a programok, amelyek az operációs rendszerrel való kölcsönhatásukat a standard könyvtáron keresztül bonyolítják, minden változtatás nélkül átvihetők az egyik számítógépről a másikra.

A könyvtári függvények tulajdonságait több mint egy tucat header állomány specifikálja, amelyek közül néhánnyal (`<stdio.h>`, `<string.h>`, `<ctype.h>`) már találkoztunk. Most nem a teljes könyvtárat fogjuk ismertetni, hanem számos érdekes C nyelvű programot írunk a könyvtári függvények felhasználásával. Magának a könyvtárnak a részletesebb leírása a B. Függelékben található.

7.1. A standard adatbevitel és adatkivitel

Amint azt az 1. fejezetben már elmondtuk, a könyvtár adatátvitelt kezelő része a szöveges adatbevitelre és adatkivitelre kialakított egyszerű modell alapján működik. A modell szerint a szövegáram egymást követő sorokból áll és minden sor egy újsor-karakterrel zárul. Ha a rendszer nem működik másképpen, akkor a könyvtári eljárás feladata annak eldöntése, hogy mi a teendő az újsor-karakter megjelenésekor. Például a könyvtári eljárás a bemenetről érkező kocsivissza- és soremelés-karaktereket egy újsor-karakterré alakítja, majd kiírás esetén elvégzi a visszaalakítást.

A legegyszerűbb adatbeviteli mechanizmus, hogy egy időben egy karaktert olvasunk be a *standard bemeneti eszközről* (szokásos módon a billentyűzetről) a `getchar` függvénnyel:

```
int getchar(void)
```

A `getchar` függvény minden hívásakor visszatér a következő karakterrel vagy az `EOF` jellel, ha az állomány végét érzékelte. Az `EOF` szimbolikus állandó az `<stdio.h>` headerben van definiálva. Az `EOF` értéke általában `-1`, de az ellenőrzéseknél inkább a szimbolikus állandót használjuk, hogy a program függetlenné váljék az adott számértéktől.

A legtöbb környezetben egy állomány helyettesítheti a billentyűzetet a megállapodás szerinti < átirányítási jelet használva. Ha a `prog` program a `getchar` függvényt használja, akkor a

```
prog <allomanyban
```

parancssor hatására a rendszer a karaktereket az `allomanyban` nevű adatállományból fogja beolvasni. A bemeneti eszköz átkapcsolása úgy történik, hogy a `prog` maga nem érzékeli a változást, az `"<allomanyban"` karaktersorozat nem kerül be az `argv` parancssor-argumentumba. A bemenet átirányítása szintén nem érzékelhető a prog számára, ha a bemeneti adatait egy másik programtól az ún pipeing mechanizmussal (láncolással) kapja. Sok rendszerben ez a

```
masprog | prog
```

parancssorral kérhető. Ennek hatására a `masprog` standard kimenete, mintegy csővezetéken keresztül rákapcsolódik a `prog` bemenetére.

A standard kimenetet az

```
int putchar(int)
```

függvény állítja elő. A `putchar(c)` a `c` karaktert adja a *standard kimeneti* eszközre, ami alapfeltételezés szerint a képernyő. A putchar függvény a hívása után a kiírt karakterrel vagy ha valamilyen hiba fordult elő, akkor az EOF jelzéssel tér vissza. A standard kimenet is átirányítható egy adatállományba a *>állománynév* parancskiegészítéssel. Ha a prog használja a `putchar` függvényt, akkor az átirányítás a

```
prog  >kiallomany
```

parancssorral történhet, és ennek hatására a standard kimenet helyett a `kiallomany` nevű adatállományba íródik a kimenet. A pipeing mechanizmus szintén megvalósítható a

```
prog | masprog
```

parancssorral, ami a `prog` standard kimenetét a `masprog` standard bemenetére irányítja.

A `printf` függvény szintén a standard kimenetet használja. A `putchar` és `printf` függvények hívásai vegyesen is előfordulhatnak és a kimenet a hívások sorrendjében jön létre.

Minden forrásállományban, amely hivatkozik az adatbeviteli–adatkiviteli könyvtár függvényeire, kell hogy szerepeljen az

```
#include <stdio.h>
```

sor az első függvényhivatkozás előtt. Amikor a rendszer megtalálja a hegyes zárójelek között a header nevét, akkor azt standard könyvtárban (pl. UNIX operációs rendszer esetén a `/usr/include` könyvtárban) kezdi keresni.

Nagyon sok program csak egyetlen bemeneti adatáramot használ és csak egyetlen kimeneti adatáramot hoz létre. Az ilyen programok adatbeviteli és adatkiviteli feladatainak ellátására a `getchar, putchar` és `printf` függvények teljesen elegendőek, és ez az induláshoz szintén elegendő. Ez különösen igaz, ha kihasználjuk az átirányítás lehetőségét, amellyel az egyik program kimenetét a másik bemenetéhez kapcsoljuk. Példaként vizsgáljuk meg a karakteres adatbevitelt és adatkivitelt használó `lower` programot, amely a bemenetére adott szöveget kisbetűs szöveggé alakítja.

```
#include <stdio.h>
```

```
#include <ctype.h>

main () /* a bemenetet kisbetűssé alakítja */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

A `tolower` függvény a `<ctype.h>` headerben van definiálva és a nagybetűket kisbetűkké alakítja, más karakterekkel érintetlenül visszatér a hívó függvénybe. Mint korábban már említettük, az `<stdio.h>` `getchar` és `putchar`, valamint a `<ctype.h>` `tolower` „függvényei" gyakran makróként vannak megvalósítva, mivel így az egy karakterre eső műveletszám (és így a futási idő is) csökkenthető. Azt, hogy ez hogyan valósítható meg, a 8.5. pontban fogjuk bemutatni. Függetlenül attól, hogy egy adott gépen ezek a függvények hogyan vannak megvalósítva, a program használja azokat és semmiféle ismerettel nem kell rendelkeznie a karakterkészletről.

7.1. gyakorlat. Írjunk programot, amely a hívásakor az `argv[0]`-ban elhelyezett paramétertől függően a nagybetűket kisbetűvé vagy a kisbetűket nagybetűvé alakítja!

7.2. A formátumozott adatkivitel – a printf függvény

A `printf` kimeneti függvény a gépen belüli értékeket karakterekké alakítja. A `printf` függvényt már a korábbi fejezetekben is használtuk és ott megadtuk a tipikus alkalmazásokra vonatkozó, de korántsem teljes leírását. A teljes leírás a B. Függelékben található. A függvény alakja:

```
int printf(char *format, arg1, arg2, ...)
```

A `printf` függvény a `format` vezérlése alatt konvertálja, formátumozza és a standard kimenetre írja az argumentumai értékét, majd visszatéréskor megadja a kiírt karakterek számát.

A `format` karaktersorozata kétféle objektumot tartalmaz: közönséges (nyomtatható) karaktereket, amelyek közvetlenül átmásolódnak a kimeneti adatáramba és konverziós specifikációkat, amelyek mindegyike a `printf` soron következő argumentumának konverzióját és kiírását eredményezi. Mindegyik konverziós specifikáció a `%` jellel kezdődik és egy konverziós karakterrel végződik. A `%` jel és a konverziós karakter között sorrendben a következők lehetnek:

- Egy mínusz jel, ami a konvertált argumentum balra igazítását írja elő.
- Egy szám, ami megadja a minimális mezőszélességet. A konvertált argumentum legalább ilyen széles mezőbe fog kiíródni. Ha szükséges, akkor a mező bal széle (vagy ha balra igazítást írtunk elő, akkor a jobb széle) üres helyekkel fog feltöltődni.
- Egy pont, ami elválasztja a mezőszélességet a pontosságot megadó számtól.
- Egy szám (pontosság), ami megadja egy karaktersorozatból kiírt karakterek maximális számát, vagy lebegőpontos adat kiírásánál a tizedespont után kiírt számjegyek számát, vagy egész típusú adat esetén a kiírt számjegyek minimális számát.
- Egy `h` betű, ha egy egész számot short típusúként vagy egy `l` betű, ha `long` típusúként írunk ki.

7.1. táblázat. A printf függvény konverziós karakterei

A konverziós karakter	Az argumentum típusa	A nyomtatás módja
d, i	int	decimális szám
o	int	előjel nélküli oktális szám vezető nullák nélkül)
x, X	int	előjel nélküli hexadecimális szám (a vezető 0x vagy 0X nélkül), a 10...15 jelzése az abcdef vagy ABCDEF karakterekkel
u	int	előjel nélküli decimális szám
c	int	egyetlen karakter
s	char *	karaktersorozatból karaktereket nyomtat a '\0' végjelzésig vagy a pontossággal megadott darabszámig
f	double	[-]m.dddddd alakú decimális szám, ahol d számjegyeinek számát a pontosság adja meg (alapfeltételezés szerint d=6)
e, E	double	[-]m.dddddde xx vagy [-]m.ddddddE xx alakú decimális szám, ahol d számjegyeinek számát a pontosság adja meg (alapfeltételezés szerint d=6)
g, G	double	%e vagy %E alakú kiírást használ, ha a kitevő < -4 vagy >= pontosság, különben a %f alakú kiírást használja. A tizedespont és az utána következő értéktelen nullák nem íródnak ki
p	void *	mutató a géptől függő kiírási formában
n	int *	a printf függvény aktuális hívásakor kiírt karakterek száma beíródik az argumentumba. Az argumentum nem konvertálódik
%		nincs konvertálandó argumentum, egy % jelet ír ki

A konverziós karaktereket a 7.1. táblázat tartalmazza. Ha a % jel utáni karakter nem konverziós specifikáció akkor a printf viselkedése definiálatlan.

A szélesség vagy pontosság a * jellel is megadható, és ebben az esetben az érték a következő argumentum (amely kötelezően int típusú kell, hogy legyen) konverziójakor jön létre. Például ha az s karaktersorozatból legfeljebb max számú karaktert akarunk kiírni, akkor ez a

```
printf("%.*s", max, s);
```

utsítással érhető el.

A formátumkonverziók többségét a korábbi fejezetekben már bemutattuk, az egyetlen kivételt a karaktersorozatok kinyomtatásánál megadott pontosság hatásának elemzése képezi. A következő példasoron bemutatjuk a pontosság előírásának hatását a

„Halló mindenki!" 15 karakteres karaktersorozat kiírására. Az egyes mezők két szélén kettőspontokat íratunk ki, hogy jobban kiemeljük a mezőt.

```
:%s:              :Halló mindenki!:
:%10s:            :Halló mindenki!:
:%.10s:           :Halló mind:
:%-10s:           :Halló mindenki!:
:%.20s:           :Halló mindenki!:
:%-18s:           :Halló mindenki!  :
:%18.10s:         :      Halló mind:
:%-18.10s:        :Halló mind       :
```

Figyelem! A `printf` függvény az első argumentumát használja annak eldöntésére, hogy még hány argumentum következik és azoknak mi a típusa. Programhiba keletkezik és hibás kiírást kapunk, ha nincs elegendő argumentum vagy azok nem a megfelelő típusúak. Ezt jól szemlélteti az alábbi két `printf` hívás összehasonlítása.

```
printf(s); /* hibás, ha s % jelet is tartalmaz */
printf("%s", s); /* ez így biztonságos */
```

A `sprintf` függvény ugyanazt a konverziót hajtja végre, mint a `printf`, de a kimenetet egy karaktersorozatban tárolja. A függvény:

```
int sprintf(char *string, char *format, arg1, arg2, ...)
```

A `sprintf` függvény először a `format` formátummegadás szerint formatálja az *arg1*, *arg2* stb. argumentumokat, majd az eredményt a standard kimenet helyett a `string` karaktersorozatba helyezi. A `string` karaktersorozatnak elegendően nagynak kell lenni ahhoz, hogy az eredményt tárolni tudja. Az `sprintf` függvényt főleg arra használhatjuk, hogy részekből egy átmeneti tárolóban állítsuk össze a kiírandó információt, majd amikor minden rész a rendelkezésünkre áll, akkor ezt az átmeneti változót a `printf` függvénnyel kiíratjuk.

7.2. gyakorlat. Írjunk programot, amely a tetszőleges bemeneti szöveget értelmes módon írja ki! A minimális igény, hogy a nem nyomtatható karaktereket a helyi szokásoknak megfelelően oktális vagy hexadecimális számként írjuk ki és a túl hosszú sorokat tördeljük rövidebb sorokra!

7.3. A változó hosszúságú argumentumlisták kezelése

Ebben a pontban megírjuk a `printf` függvény minimális igényeket kielégítő változatát, hogy bemutassuk, hogyan lehet olyan hordozható függvényeket írni, amelyek képesek egy változó hosszúságú argumentumlista feldolgozására. Mivel főleg az argumentumok feldolgozására helyeztük a hangsúlyt, az általunk írt `minprintf` függvény csak a formátumot leíró karaktersorozatot és az argumentumokat dolgozza fel, a formátumkonverziót a valódi `printf` függvény hívásával valósítja meg.

A `printf` deklarációja:

```
int printf(char *fmt, ... )
```

ahol a ... azt jelzi, hogy az argumentumok száma és típusa változhat. A deklarációban a ... csak az argumentumlista végén jelenhet meg. Az általunk írt `minprintf` függvény deklarációja

```
void minprintf(char *fmt, ...)
```

mivel mi nem adjuk vissza a kiírt karakterek számát, mint a `printf`.

Egy kicsit „trükkös", ahogyan a `minprintf` végigmegy az argumentumlistán, miközben az egyetlen nevet sem tartalmaz. A `<stdarg.h>` headerben néhány makrodefiníció található, amelyek lehetővé teszik az argumentumlistán való lépkedést. Ennek a headernek a konkrét kialakítása számítógéptől függ, de a makrók szoftver-interfésze egységes.

A `va_list` típust fogjuk használni a sorjában következő egyes argumentumokra hivatkozó változók deklarálására. A `minprintf` függvényben ezt a változót `am`-nek nevezzük, az „argumentummutató" kifejezés alapján. A `va_start` makró úgy inicializálja az `am` változót, hogy az az első név nélküli argumentumra mutasson. Ezért `am` használata előtt a `va_start` makrót egyszer végre kell hajtatni. A listában legalább egy névvel ellátott argumentumnak kell lenni és ezt az utolsó, névvel ellátott argumentumot használja a `va_start` a meg nem nevezett argumentumok listáján való elinduláshoz.

A `va_arg` minden egyes hívása után egy argumentummal tér vissza és az `am` mutatót tovább lépteti a következő argumentumra. A `va_arg` visszatérésekor az argumentum értéke olyan típusú lesz, mint amit hívásakor megadtunk. Az argumentumlista feldolgozási folyamatát a `minprintf` függvényt záró `return` utasítás kiadása előtt a `va_end` makró hívásával kell lezárni. (Az argumentumlistát feldolgozó makrók leírása a B. Függelék 7. pontjában található.)

Ezekkel a jellemzőkkel kialakított `minprintf` függvény:

```
#include <stdarg.h>

/* minprintf: változó hosszúságú argumentumlistát
kezelő, egyszerűsített printf függvény */
void minprintf(char *fmt, ...)
{
    va_list am; /* sorjában az egyes név nélküli
          argumentumokra mutat */
    char *p, *sert;
    int iert;
    double dert;

    va_start(am, fmt); /* hatására am az első név
          nélküli argumentumra fog mutatni */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            iert = va_arg(am, int);
            printf("%d", iert);
            break;
```

```
            case 'f':
                dert = va_arg(am, double);
                printf("%f", dert);
                break;
            case 's':
                for (sert = va_arg(am, char *); *sert; sert++)
                    putchar(*sert);
                break;
            default:
                putchar(*p);
                break;
            }
        }
        va_end(am); /* a listafeldolgozás lezárása */
    }
```

7.3. gyakorlat. Egészítsük ki a `minprintf` függvényt újabb, `printf` függvényben megengedett lehetőségekkel!

7.4. Formátumozott adatbevitel – a scanf függvény

Az adatbevitelt végző `scanf` függvény a `printf` analógiája, és többségében azonos (de ellentétes irányú) adatkonverziókat képes elvégezni. Általános alakja:

```
int scanf(char *format, ...)
```

A `scanf` függvény a standard bemenetről karaktereket olvas és a `format`-ban megadott specifikációk szerint értelmezi azokat, majd az eredményt eltárolja a további argumentumokban. A formátumot leíró argumentumot hamarosan részleteiben is tárgyaljuk, a többi argumentumnak viszont *kötelezően mutatónak kell lenni,* ami arra a helyre mutat, ahová a konvertált értéket helyezzük. Csakúgy, ahogyan a `printf` esetén, itt is csak a `scanf` legfontosabb jellemzőit írjuk le és nem megyünk bele a részletekbe.

A `scanf` függvény működése befejeződik, ha feldolgozta a formátumot leíró karaktersorozatot vagy hibát érzékel (az aktuális bemenet nem illeszkedik a vezérlő specifikációhoz). A függvény visszatérési értéke a sikeresen illesztett (konvertált) és argumentumokban elhelyezett adatok száma. Ebből eldönthető, hogy a `scanf` hány bemeneti tételt talált. Az adatállomány végének elérésekor a visszaadott érték `EOF`. Ügyeljünk arra, hogy ez egy nullától különböző érték és azt jelenti, hogy a következő bemeneti karakter nem illeszkedik a formátumot leíró karaktersorozat első specifikációjához! Ha a `scanf` függvényt többször egymás után hívjuk, akkor a következő hívás esetén a keresés közvetlenül az utoljára visszaadott és már konvertált karakter után folytatódik.

A `scanf` függvénynek létezik egy `sscanf` változata, ami a `sprintf`-hez hasonló és a standard bemenet helyett egy karaktersorozatból olvas:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

A `sscanf` függvény a `format`-ban megadott formátum szerint kiolvassa a `string`

karaktersorozatot és a konvertált értékeket elhelyezi az *arg1*, *arg2* stb. argumentumokban. Természetesen ezek az argumentumok is mutatók.

A formátumot leíró karaktersorozat konverziós specifikációkat tartalmaz, amelyeket a bemenet átalakításának vezérlésére használunk. A formátumvezérlő karaktersorozat a következő karaktereket tartalmazhatja:

- Szóközök és tabulátorkarakterek, amelyeket a `scanf` nem vesz figyelembe.
- Közönséges (% jeltől különböző) karakterek, amelyek várhatóan illeszkednek a bemeneti adatáram következő nem üres karakteréhez.
- Konverziós specifikáció, ami a `%` jelből, az opcionális hozzárendelés-elnyomó * karakterből, a maximális mezőszélességet előíró opcionális számból, egy opcionális `h`, `l` vagy `L` betűből (amelyek a célmező szélességét jelzik), valamint egy konverziós karakterből áll.

A konverziós specifikáció irányítja a következő bemeneti mező átalakítását. Normális esetben az eredmény a megfelelő argumentummal (mint mutatóval) címzett vátlozóba kerül. Ha a hozzárendelés-elnyomó * karaktert alkalmaztuk, akkor a `scanf` a bemeneti mezőt átlépi és nem történik meg az érték változóhoz rendelése. Egy bemeneti mező alatt a nem üres karakterekből álló karaktersorozatot értjük, ami a következő üres karakterig tart, vagy addig, amíg a mezőszélesség (ha megadtuk) el nem fogy. Ebből következik, hogy a `scanf` átmegy a sorhatárokon is a bemeneti adat keresése közben, mivel az újsor-karakter is üres karakternek számít. (Üres karakter a szóköz, a tabulátor, a kocsi-vissza, a soremelés, a függőleges tabulátor és a lapdobás.)

A konverziós karakter adja meg a bemeneti mező értelmezését. A neki megfelelő argumentumnak mutatónak kell lenni, ahogyan ezt a C nyelv érték szerinti hívása megköveteli. A konverziós karaktereket a 7.2. táblázat tartalmazza.

A `d`, `i`, `o`, `u` és `x` konverziós karakterek előtt álló `h` azt jelzi, hogy a mutató az argumentumlistában szereplő `int` típus `short` változatára mutat, és hasonlóan az `l` a `long` változatra utal. Az `e`, `f` és `g` konverziós karakterek előtt megjelenő `l` arra utal, hogy a `float` típusú argumentum `double` változatú.

A `scanf` használatának bemutatását kezdjük a 4. fejezetben ismertetett egyszerű kalkulátor programjának módosításával! A programban a bemeneti adatok átalakítását oldjuk meg a `scanf` függvénnyel.

```
#include <stdio.h>

main() /* egyszerű kalkulátor */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

A következő példában tegyük fel, hogy

```
1994 december 25
```

alakban írt dátumot akarunk beolvasni. Ez a `scanf` függvénnyel

```
int nap, ev;
```

7.2. táblázat. A scanf függvény konverziós karakterei

A konverziós karakter	Az argumentum típusa	A beolvasott adat
d	int *	decimális egész
i	int *	egész szám, ami lehet oktális (vezető nullákkal) vagy hexadecimális (vezető 0x vagy 0X karakterekkel)
o	int *	oktális egész szám (vezető nullákkal vagy azok nélkül)
u	unsigned int *	előjel nélküli decimális egész szám
x	int *	hexadecimális egész szám (a vezető 0x, ill. 0X karakterekkel vagy azok nélkül)
c	char *	karakterek. A következő bemeneti karakterek (alapfeltételezés szerint 1) elhelyezése a kijelölt mezőben. Az üres helyek átlépését (mint normális esetet) elnyomja, ha a következő nem üres karaktert akarjuk beolvastatni, akkor a %1s specifikációt kell használni
s	char *	karaktersorozat (aposztrófok nélkül). A char * mutató egy elegendően nagy karaktersorozatra mutat és a záró '\0' jelzést a beolvasás után automatikusan elhelyezi
e, f, g	float *	lebegőpontos szám, opcionális előjellel opcionális tizedesponttal és opcionális kitevővel
p	void *	mutató, olyan formában, ahogyan azt a printf ("%p") kiírta
n	int *	az aktuális scanf hívással beolvasott karakterek száma beíródik az argumentumba. Nem történik adatbeolvasás, a konvertált tételek száma nem nő
[...]	char *	a bemeneti karakteráramból beolvassa a zárójelek közötti karakterekkel (illeszkedési halmazzal) megegyező karakterekből álló leghosszabb nem üres karaktersorozatot és lezárja a '\0' végjellel. A [] . . .] formában megadott halmaz esetén a] karakter a halmaz része lesz
[^...]	char *	az illeszkedési halmazzal nem megegyező karakterekből álló karaktersorozat beolvasása és '\0' végjellel történő lezárása. A [^] . . .] formában megadott halmaz esetén a] karakter a halmaz része lesz
%		nincs hozzárendelés, a % jel mint karakteres állandó

```
char honapnev[20];

scanf ("%d %s %d", &ev, honapnev, &nap);
```

formában valósítható meg a beolvasás. Vegyük észre, hogy a honapnev előtt nem írtuk ki az & jelet, mivel egy tömbnév mindig mutató!

A scanf formátumot leíró karaktersorozatában karakteres állandók (literálisok) is megjelenhetnek, de azoknak illeszkedni kell a bemenetről érkező ugyanolyan karakterekhez. Ha pl. a dátumot éé/hh/nn alakban akarjuk beolvastatni a scanf függvénnyel, akkor az

```
int nap, honap, ev;

scanf ("%d/%d/%d", &ev, &honap, &nap);
```

formátumleírást kell alkalmazni.

A scanf függvény beolvasáskor kiszűri a formátumot megadó karaktersorozatban lévő szóközöket és tabulátorokat, sőt átlépi az üreshely-karaktereket (szóköz, tabulátor, új sor stb.) a beolvasott adatáramban is a beolvasandó érték keresése közben. Ha a bemeneti adatáram nem rögzített formátumú, akkor célszerű egyszerre egy egész adatsort beolvasni és az sscanf függvénnyel részleteiben elővenni és feldolgozni. Ha például olyan sort akarunk beolvasni, ami a korábbi két dátumírási mód bármelyikével írt dátumot tartalmaz, akkor ezt úgy tehetjük meg, hogy a getline függvénnyel beolvassuk a teljes sort, majd az sscanf függvénnyel feldolgozzuk.

```
while (getline(sor, sizeof(sor)) > 0) {
    if (sscanf(sor, "%d %s %d", &ev, honapnev,&nap) == 3)
        printf ("érvényes: %s\n", sor);
            /* 1994 december 25 alakú dátum */
    else if (sscanf(sor, "%d/%d/%d", &ev, &honap,
            &nap) == 3)
        printf("érvényes: %s\n", sor);
            /* éé/hh/nn alakú dátum */
    else
        printf("érvénytelen: %s\n", sor);
            /* érvénytelen alakú dátum */
}
```

A scanf és más adatbeviteli függvények hívásai egymással keverhetők. Bármelyik adatbeviteli függvény következő hívásakor a beolvasás az első, scanf által már be nem olvasott karakterrel kezdődik.

Befejezésül még egyszer kihangsúlyozzuk, hogy a scanf és sscanf *függvények argumentumainak mutatóknak kell lennie!* Nagyon gyakori hiba, hogy

```
scanf("%d", n); /* HIBÁS!!! */
```

alakban írjuk a függvényhívást a

```
scanf("%d", &n); /* Helyes! */
```

helyett. Ez a hiba általában nem derül ki a program fordítása közben.

7.4. gyakorlat. Írjuk meg a `scanf` függvény egyszerűsített változatát az előző pontban látott `minprintf` mintájára!

7.5. gyakorlat. Írjuk meg a 4. fejezetben ismertetett postfix adatbeírási formátumú kalkulátorprogram új változatát, amely a bemeneti adatok és számok átalakítását a `scanf` és/vagy `sscanf` függvénnyel valósítja meg!

7.5. Hozzáférés adatállományokhoz

Az eddigi példaprogramok mindegyike a standard bemenetről olvasta az adatokat és a standard kimenetre írta az eredményt. A standard bemenetet és kimenetet a helyi operációs rendszer automatikusan hozzárendeli a programhoz.

A következőkben egy olyan programot mutatunk be, amely egy adatállományhoz fér hozzá, amit *nem* az operációs rendszer rendelt átirányítással a programhoz. Az adatállományokhoz való hozzáférés szükségességét és megvalósítási lehetőségét illusztráló `cat` program megnevezett adatállományok halmazát gyűjti egybe (konkatenálja az állományokat) és az eredményt a standard kimenetre írja. A `cat` fő alkalmazási területe, hogy állományokat gyűjtsön egybe és nyomtassa ki azokat, vagy az önálló, név szerinti állomány-hozzáférésre nem felkészített programok bemeneti információit gyűjtse be. Például a

```
cat x.c y.c
```

parancs a standard kimenetre írja az `x.c` és `y.c` nevű állományok (és csak ezen állományok) tartalmát.

A program kialakításánál a fő kérdés, hogy hogyan érhetjük el a megnevezett állományok beolvasását, vagyis azt, hogy a felhasználó által szabadon választott külső állománynevet az adatbeolvasó utasításhoz rendeljük.

A szabály rendkívül egyszerű! Mielőtt az állományból olvasnánk vagy abba írnánk, az állományt a `fopen` könyvtári függvénnyel meg kell nyitni. A `fopen` veszi a külső állománynevet (mint pl. `x.c` vagy `y.c`), mindenféle belső adminisztrációt végez, felveszi a kapcsolatot az operációs rendszerrel (amelynek részleteivel nem kell törődnünk), majd egy mutatót ad vissza, ami ezt követően az állomány olvasásánál vagy írásánál használható.

Ez a mutató, amit *állománymutatónak (fájlpointernek)* neveznek, egy struktúrát címez, ami az állományra vonatkozó információkat (a puffer helye; az aktuális karakterpozíció a pufferban; annak jelzése, hogy az állományt írásra vagy olvasásra veszszük-e igénybe; a hiba vagy állományvég előfordulásakor szükséges teendők stb.) tartalmazza. A felhasználónak nem szükséges a részleteket ismerni, mivel a struktúra `FILE` néven definiálva van az `<stdio.h>` standard headerben. Az állománykezeléshez csak az állománymutatót kell deklarálni a következő minta szerint:

```
FILE *fp;
FILE *fopen(char *nev, char *mod);
```

A deklaráció azt mondja ki, hogy `fp` egy `FILE` típusú struktúrához tartozó mutató és a `fopen` egy `FILE` típusú struktúrát címző mutatóval tér vissza. Vegyük észre, hogy `FILE` egy típusnév, mint pl. az `int`, és nem igazi struktúranév. A `FILE` típust `typedef` utasítással deklarálták a könyvtárban (egyébként a `fopen` UNIX operációs rendszer alatti megvalósításának részleteit a 8.5. pontban írjuk le). A `fopen` függvény egy programból az

```
fp = fopen(nev, mod);
```

utasítással hívható. A fopen első argumentuma egy karaktersorozat, amely a megnyitandó állomány nevét tartalmazza. A második argumentum szintén egy karaktersorozat, ami azt jelzi, hogy a felhasználó hogyan akarja használni az állományt (használati mód). A használati mód karaktersorozatában használható karakterek: olvasás ("r"), írás ("w") és hozzáfűzés ("a", append). Néhány rendszer különbséget tesz szöveges és bináris állományok között, ilyenkor bináris állomány esetén a használati módot a "b" karakterrel kell kiegészíteni.

Ha írásra vagy hozzáfűzésre egy nem létező állományt akarunk megnyitni, akkor az adott nevű állomány (ha lehetséges) létrejön. Egy létező állományt írásra megnyitva, annak korábbi tartalma elvész, hozzáfűzésre megnyitva viszont megmarad. Ha megpróbálunk olvasásra megnyitni egy nem létező állományt, akkor hibajelzést kapunk. Más hibaokok is előfordulhatnak, pl. ha olvasni akarunk egy létező, de számunkra nem hozzáférhető (nem engedélyezett) állományt. Ha bármilyen állománykezelési hiba fordul elő, akkor a fopen NULL értékű mutatóval tér vissza. A hiba oka pontosabban is meghatározható, az erre alkalmas hibakezelő függvények leírása a B. Függelék 1. részében található.

A cat program megírásához szükséges következő tudnivaló, hogy hogyan lehet a már megnyitott állományt olvasni vagy írni. Ennek számos lehetősége van, amelyek közül a legegyszerűbb a getc és putc függvények alkalmazásán alapszik. A getc függvény az állományból kiolvasott következő karakterrel tér vissza, és hívásakor az állománymutató megadásával kell közölni, hogy melyik állományból olvasson. Általános alakja:

```
int getc(FILE *fp)
```

A függvény az fp-vel címzett adatáramból vett következő karakterrel vagy hiba esetén EOF jelzéssel tér vissza.

A putc adatkiviteli függvény

```
int putc(int c, FILE *fp)
```

alakú, és hívásakor a függvény az fp-vel címzett állományba kiírja a c karaktert, és visszatér a kiírt karakterrel vagy hiba esetén az EOF jellel. Hasonlóan a getchar és putchar eljárásokhoz, gyakran a getc és putc eljárásokat is makrók formájában írják meg, nem pedig függvényként, de ez a használatukat nem befolyásolja.

Egy C nyelvű program indításakor az operációs rendszer három állományt mindig automatikusan megnyit és az ezekhez tartozó állománymutatókat a program rendelkezésére bocsátja. Ez a három állomány a standard bemenet, a standard kimenet és a standard hibaállomány, amelyek állománymutatói a <stdio.h> headerben vannak deklarálva stdin, stdout és stderr néven. Normális esetben az stdin a billentyűzethez, stdout a képernyőhöz kapcsolódik, de ahogy azt a 7.1. pontban már leírtuk, az stdin és stdout minden további nélkül átirányítható adatállományokhoz vagy a pipeing mechanizmussal egy másik programhoz.

A getc, putc, stdin és stdout felhasználásával a getchar és putchar a következő módon definiálható:

```
#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)
```

Adatállományok formátumozott olvasására vagy írására az fscanf és fprintf függvényeket használhatjuk. Ezek használata lényegében megegyezik a scanf vagy printf függvények használatával, kivéve, hogy az első argumentumuknak az olvasandó vagy

írandó állományt kijelölő állománymutatónak kell lennie és a formátumot leíró karaktersorozat a második argumentum. A függvények általános alakja:

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Ezen előzetes ismeretek birtokában most már hozzákezdhetünk az állományokat összekapcsoló `cat` program írásához! A program felépítése megegyezik a korábbi programoknál már bevált felépítéssel: ha a programot parancssor-argumentummal hívjuk, akkor az argumentumok az állománynevek, ha nincs argumentum, akkor a standard bemenetről jövő adatok kerülnek feldolgozásra. A program:

```
#include <stdio.h>

/* cat: állományok konkatenálása - 1. változat */
main (int argc, char *argv[])
{
    FILE *fp;
    void filecopy (FILE *, FILE *);

    if (argc == 1) /* nincs argumentum, a standard
            bemenetet másolja */
        filecopy (stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                printf("cat: nem nyitható meg %s\n", *argv);
                return 1;
            } else {
                filecopy(fp, stdout);
                fclose(fp);
            }
    return 0;
}

/* filecopy: az ifp-vel címzett állományt
az ofp-vel címzett állományba másolja */
void filecopy (FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
        putc (c, ofp);
}
```

Az `stdin` és `stdout` állománymutatók `FILE *` típusú objektumok. Az `stdin` és `stdout` állandók, amelyeket a standard könyvtárban definiáltak, és *nem* pedig változók, így értéket sem rendelhetünk hozzájuk. Az

```
int fclose(FILE *fp)
```

függvény a fopen inverze, és hatására megszakad a kapcsolat az állománymutató és a külső állománynév között, az állománymutató felszabadul és más állományhoz rendelhető. A legtöbb operációs rendszer korlátozza az egy program futása során egyidejűleg nyitott állományok számát, ezért amikor már nincs szükség rá, célszerű felszabadítani az állománymutatót, ahogy ezt a cat programban is tettük. Az fclose másik fontos feladata, hogy kimeneti állományok esetén kiírja az állományba a putc által használt puffer tartalmát. (Normális esetben a puffer tartalma csak akkor íródna ki, ha már a puffer megtelt. Az utolsó, általában nem egész puffernyi tartalmat az fclose függvénnyel kell kiíratni.) A fclose automatikusan végrehajtódik minden egyes megnyitott állományra, ha a program normálisan fut le. A standard bemenetet és kimenetet szintén lezárhatjuk, ha nincs szükségünk rájuk. Amennyiben újra szükségessé válik a használatuk, akkor a freopen könyvtári függvénnyel nyithatók meg újra.

7.6. Hibakezelés – az stderr és exit függvények

A cat program hibakezelése messze nem ideális. A problémát az okozza, hogy ha egy állomány valamilyen hiba folytán hozzáférhetetlenné válik, a hibaüzenet a konkatenált kimeneti állomány (vagyis az eredmény) végére íródik ki. Ez megfelelő lehet, ha a kimeneti eszköz a képernyő, viszont elfogadhatatlan, ha egy állományba írunk vagy az eredményt a pipeing mechanizmussal egy másik programnak adjuk át.

A hibák jobb kezelése érdekében a programhoz automatikusan egy második kimeneti állományt is hozzárendel az operációs rendszer (hasonlóan az stdin és stdout állományokhoz), és ez az stderr hibaállomány. Az stderr állományba írt üzenetek normális esetben mindig a képernyőn jelennek meg, még akkor is, ha a standard kimenetet átirányítottuk.

Ennek alapján módosítsuk a cat programot és a hibaüzenetet írassuk a standard hibaállományba!

```
#include <stdio.h>

/* cat: állományok konkatenálása - 2. változat */
main(int argc, char *argv[])
{
    FILE *fp;
    void filecopy(FILE *, FILE *);
    char *prog = argv[0]; /* a program neve hiba esetén */
    if (argc == 1) /* nincs parancssor-argumentum,
            a standard bemenetről másol */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "f")) == NULL) {
                fprintf(stderr, "%s: nem nyitható meg: %s\n", prog, *argv);
            exit (1);
        } else {
            filecopy(fp, stdout);
            fclose(fp);
        }
    if (ferror(stdout)) {
        fprintf(stderr, "%s: hiba stdout írásakor\n",
                prog);
        exit (2);
```

```
        }
        exit (0);
    }
```

A program a hibákat kétféle módon jelzi. Először is az `fprintf` által létrehozott hibaüzenetek az `stderr` állományba íródnak, tehát mindig a képernyőn jelennek meg, ahelyett, hogy a kimeneti (eredmény) állományba íródnának vagy a pipeing mechanizmussal egy másik program bemenetére kerülnének. A hibaüzenet kiírásába belefoglaltuk a program `argv[0]`-ból vett nevét, így ha a programot más programokkal együtt használjuk, azonosítható a hiba forrása.

Az `stderr` állományon keresztüli hibajelzésen kívül a program használja az `exit` standard könyvtári függvényt, ami hívásakor leállítja a program futását. Az `exit` argumentuma bármely, az `exit`-et hívó eljárás rendelkezésére áll, így a program sikeres vagy hibás végrehajtása egy másik, ezt a programot alprogramként használó eljárásból ellenőrizhető. Megállapodás szerint a 0 argumentum a program sikeres lefutását, a nem nulla argumentum pedig valamilyen, normálistól eltérő működését jelzi. Az `exit` függvény automatikusan hívja az `fclose` függvényt és minden egyes megnyitott kimeneti állományba kiírja a kimeneti pufferének tartalmát.

A `main` eljárásban a `return` *kifejezés* utasítás egyenértékű az `exit` (*kifejezés*) utasítással. Viszont az `exit` előnye, hogy az más függvényekből is hívható és a hívás helye az 5. fejezetben leírt mintakereső programhoz hasonló programmal meghatározható.

Az `ferror` függvény nem nulla értékkel tér vissza, ha az `fp` állománymutatóval megcímzett állomány írása vagy olvasása közben hiba fordult elő. A függvény általános alakja:

```
int ferror(FILE *fp)
```

A gyakorlatban az adatkiviteli hibák ritkábban fordulnak elő, mint a beolvasási hibák (bár pl. hibát jelent, ha lemezre írás közben elfogy az üres hely), de a programot mindenképpen célszerű ellenőrizni.

A `feof (FILE *)` függvény analóg a `ferror` függvénnyel és szintén nem nulla értékkel tér vissza, ha a megadott állományhoz való hozzáférés során hiba fordult elő. A függvény általános alakja:

```
int feof(FILE *fp)
```

A kis mintaprogramunk kapcsán általában nem érdemes túlzottan aggódni a kilépéskori állapotjelzés miatt, de nagyobb programok esetén gondot kell fordítani az értelmes és jól használható állapotjelzésekre.

7.7. Szövegsorok beolvasása és kiírása

A standard könyvtár tartalmazza az `fgets` bemeneti függvényt, ami hasonló a korábbról már ismert `getline` függvényhez, és általános alakja:

```
char *fgets(char *sor, int maxsor, FILE *fp)
```

Az `fgets` függvény az `fp` állománymutatóval megadott állományból beolvassa a következő bemenő sort (az újsor-karaktert is beleértve) és elhelyezi a `sor` nevű ka-

rakteres tömbben. A függvény legfeljebb maxsor-1 karaktert fog beolvasni és a beolvasott sort a '\0' jellel fogja lezárni. Normális esetben az fgets a sor tömbbel tér vissza, de állomány vége vagy hiba esetén a visszatérési érték NULL. (A korábban használt getline függvény a sor hosszával tért vissza, ami nagyon hasznos volt, mivel így a nulla jelenthette az állomány végét.)

Adatkiírásra az fputs függvény használható, amivel egy karaktersorozat (ami nem szükségszerű, hogy újsor-karaktert tartalmazzon) írható ki egy megadott állományba. A függvény általános alakja:

```
int fputs(char *sor, FILE *fp)
```

A függvény EOF jellel tér vissza, ha a kiíráskor hibát érzékel és nulla értékkel, ha minden rendben volt.

A könyvtári gets és puts függvények hasonlóak az fgets és fputs függvényekhez, de mindig az stdin és stdout állománymutatókkal kijelölt állományt kezelik. Használatuk során zavaró lehet, hogy a gets törli a billentyűzetről érkező '\n' újsor-karaktert, a puts pedig mindig azzal zárja a kiírást.

Annak bizonyítására, hogy az fgets vagy fputs függvényekben nincs semmi speciális, idemásoltuk a könyvtári függvények eredeti programkódját:

```
/* fgets: beolvas egy legfeljebb n karakteres sort
az iop állománymutatóval kijelölt állományból */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\n';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs: kiírja az s karaktersorozatot
az iop állománymutatóval kijelölt állományba */
int fputs(char *s, FILE *iop)
{

    int c;

    while (c = *s++)
        putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

A szabvány azt írja elő, hogy a ferror függvény nem nulla értékkel tér vissza, ha hiba volt, ezzel szemben az fputs hiba esetén EOF jelzéssel, minden más esetben nem negatív értékkel tér vissza.

Az fgets függvény felhasználásával már egyszerűen megvalósíthatjuk a getline függvényt:

```
/* getline: beolvas egy sort és visszatér
a sor hosszával */
int getline(char *sor, int max)
{
    if (fgets(sor, max, stdin) == NULL)
        return 0;
    else
        return strlen(sor);
}
```

7.6. gyakorlat. Írjunk programot, amely összehasonlít két állományt, és kiírja az első olyan sort, ahol különböznek!

7.7. gyakorlat. Módosítsuk az 5. fejezetben ismertetett mintakereső programot úgy, hogy bemenetét a parancssor-argumentumként megadott állománynevek listájából, vagy ha nincs argumentum, akkor a standard bemenetről vegye! Ki kell-e íratni az állomány nevét, ha a program egymáshoz illeszkedő sorokat talál?

7.8. gyakorlat. Írjunk programot, amely több állományt ír ki egymás után, minden állományt új oldalon kezdve! A program minden állomány kiírása előtt írja ki a lap tetejére a címet és az oldalakat állományonként folyamatosan számozza!

7.8. További könyvtári függvények

A standard könyvtárban számos, különböző feladatok megoldására alkalmas függvény található. Ebben a pontban néhány hasznos függvény rövid leírását adjuk. A részletes leírás, ill. a könyvtár további függvényeinek ismertetése a B. Függelékben található.

7.8.1. Karaktersorozatokat kezelő függvények

Mint már korábban említettük, az strlen, strcpy, strcat, ill. strcmp függvények deklarációja a <string.h> standard headerben található, a többi, karaktersorozatot kezelő függvény deklarációjával együtt. A következő leírásban s és t char * típusú karaktersorozatot, c és n int típusú adatot jelöl. Az egyes karaktersorozat-kezelő függvények:

strcat(s, t)	a t karaktersorozatot az s végéhez fűzi (konkatenálja);
strncat(s, t, n)	a t karaktersorozat n darab karakterét az s végéhez fűzi;
strcmp(s, t)	negatív, nulla vagy pozitív értékkel tér vissza, ha s < t, s = t vagy s > t;
strncmp(s, t, n)	ugyanúgy működik, mint az strcmp, de az összehasonlítást csak az első n karakterrel végzi el;
strcpy(s, t)	a t karaktersorozatot s-be másolja;
strncpy(s, t, n)	a t karaktersorozat első n karakterét s-be másolja;
strlen(s)	a visszatérési érték az s karaktersorozat hossza;
strchr(s, c)	visszatér a c karakter s karaktersorozatbeli első előfordulási helyének mutatójával, vagy NULL értékkel, ha c nem fordul elő s-ben;

`strrchr(s, c)` visszatér a `c` karakter `s` karaktersorozatbeli utolsó előfordulási helyének mutatójával, vagy `NULL` értékkel, ha `c` nem fordul elő `s`-ben.

7.8.2. Karaktervizsgáló és -átalakító függvények

A `<ctype.h>` standard header számos, karakterek vizsgálatára vagy átalakítására alkalmas függvényt tartalmaz. A leírt függvények visszatérési értéke `int` típusú, a leírásukban szereplő `c int` típusú, és `unsigned char` vagy `EOF` adatként értelmezhető.

`isalpha(c)` visszatérési érték nem nulla, ha `c` alfabetikus karakter és nulla, ha nem az;
`isupper(c)` visszatérési érték nem nulla, ha `c` nagybetűs karakter és nulla, ha nem az;
`islower(c)` visszatérési érték nem nulla, ha `c` kisbetűs karakter és nulla, ha nem az;
`isdigit(c)` visszatérési érték nem nulla, ha `c` számjegykarakter és nulla, ha nem az;
`isalnum(c)` visszatérési érték nem nulla, ha `c` alfabetikus vagy számjegykarakter és nulla, ha nem az;
`isspace(c)` visszatérési érték nem nulla, ha `c` szóköz-, tabulátor-, újsor-, kocsivissza-, lapemelés- vagy függőlegestabulátor-karakter és nulla, ha nem az;
`toupper(c)` visszatérési érték `c` nagybetűssé alakított értéke;
`tolower(c)` visszatérési érték `c` kisbetűssé alakított értéke.

7.8.3. Az ungetc függvény

A standard könyvtárban megtalálható a 4. fejezetben megírt `ungetch` függvény egy szűkített változata, az `ungetc`. A függvény általános alakja:

```
int ungetc(int c, FILE *fp)
```

Az `ungetc` függvény a `c` karaktert visszahelyezi az `fp` állománymutatóval kiválasztott állományba, és visszatér magával a `c` értékkel, vagy hiba esetén az `EOF` jelzéssel. Állományonként csak egy karakter visszahelyezése megengedett. Az `ungetc` függvény minden `scanf`, `getc` vagy `getchar` jellegű bemeneti függvénnyel használható.

7.8.4. Parancsvégrehajtás

A `system (char *s)` függvény végrehajtja az `s` karaktersorozatban elhelyezett parancsot, ha az éppen futó programban rá kerül a vezérlés. Az `s` karaktersorozat megengedett tartalma (a megengedett parancsok halmaza) nagymértékben függ a használt operációs rendszertől. A `system` függvény alkalmazására jó példa a UNIX operációs rendszer esetén kiadott

```
system("date");
```

utasítás, ami a rendszer `date` parancsának végrehajtását, azaz a dátum és a napi idő standard kimeneten való kiírását idézi elő. A `system` a használt operációs rendszertől függő egész állapotjelzéssel tér vissza a végrehajtott parancsból. UNIX operációs rendszer esetén a `system` visszatérési állapotjelzése megegyezik az `exit` visszatérési értékével.

7.8.5. Tárkezelő függvények

A tárolóban adott méretű terület dinamikus lefoglalása a `malloc` és `calloc` függvényekkel lehetséges. A `malloc` függvény általános alakja:

```
void *malloc(size_t n)
```

A függvény egy `n` bájtos, inicializálatlan tárterületet címző mutatóval, vagy ha a helyfoglalási igény nem elégíthető ki, a `NULL` értékű mutatóval tér vissza. A

```
void *calloc(size_t n, size_t meret)
```

általános alakú `calloc` függvény `n` darab megadott méretű objektum számára elegendő helyet biztosító tömb mutatójával, vagy ha a helyigény nem elégíthető ki, akkor `NULL` értékű mutatóval tér vissza.

A `malloc` vagy `calloc` függvények visszatérési értékeként kapott mutató a megadott objektumnak megfelelő helyre mutat, de kényszerített típuskonverzióval a kívánt típusúvá kell alakítani, pl. az

```
int *ip;
ip = (int *) calloc(n, sizeof(int));
```

módon.

A `free(p)` függvény felszabadítja a `p` mutatóval megcímzett helyet, ahol `p` egy eredendően `malloc` vagy `calloc` hívásával kapott mutató. Arra vonatkozóan, hogy melyik helyet szabadítjuk fel, nincs semmiféle megszorítás, de fatális hiba a következménye, ha nem a `malloc` vagy `calloc` függvény hívásával lefoglalt helyet akarunk felszabadítani.

Szintén programhibát okoz, ha a hely felszabadítása után akarjuk használni az adott hely tartalmát. Az alábbi, egy lista helyeit felszabadító ciklus tipikus, de inkorrekt programot ad:

```
for (p = fej; p != NULL; p = p->kovetkezo) /* HIBÁS! */
    free(p);
```

A feladatot helyesen csak úgy tudjuk megoldani, ha a hely felszabadítása előtt annak tartalmát elmentjük egy segédváltozóba:

```
for (p = fej; p != NULL; p = q) {
    q = p->kovetkezo;
    free(p);
}
```

A 8.7. pontban bemutatjuk egy `malloc`-hoz hasonló tárhelykiosztó függvény megvalósítását. Az általa kiosztott tárterületek tetszőleges sorrendben szabadíthatók fel.

7.8.6. A matematikai függvények

A `<math.h>` standard headerben húsznál több matematikai függvény van deklarálva, itt most csak a leggyakrabban használatosakat mutatjuk be. Mindegyik függvénynek egy vagy két `double` típusú argumentuma van és viszatérési értéke `double` típusú.

`sin(x)`	a radiánban adott x érték szinusza;
`cos(x)`	a radiánban adott x érték koszinusza;
`atan2(y, x)`	az y/x árkusz tangense radiánban;
`exp(x)`	az e^x exponenciális függvény értéke;
`log(x)`	x természetes (e alapú) logaritmusa ($x > 0$);
`log10(x)`	x tízes alapú logaritmusa ($x > 0$);
`pow(x, y)`	az x^y függvény értéke;
`sqrt(x)`	x négyzetgyöke ($x \geq 0$);
`fabs(x)`	x abszolút értéke.

7.8.7. Véletlenszám-generálás

A `rand ()` függvény egész számok pszeudovéletlen-szerű sorozatát állítja elő. A kapott számok nulla és az `<stdlib.h>` standard headerben definiált `RAND_MAX` érték közé esnek. Ennek felhasználásával a $0 \leq x < 1$ tartományba eső lebegőpontos véletlenszámok a

```
#define frand()((double) rand() / (RAND_MAX+1.0))
```

definícióval állíthatók elő. (Ha az adott gépen futó C rendszer könyvtárában már létezik a lebegőpontos véletlenszám-generáló függvény, akkor az valószínűleg kedvezőbb statisztikai tulajdonságokkal rendelkezik, mint az így definiált véletlen szám.)

A `rand` függvény kiindulási értéke a `srand(unsigned)` függvénnyel állítható be. A `rand` és `srand` függvények szabványban javasolt hordozható változatát a 2.7. pontban ismertettük.

7.9. gyakorlat. Az `isupper`-hez hasonló függvények helytakarékos vagy időtakarékos változatban írhatók meg. Vizsgálja meg, hogyan lehetséges mindkét változat kialakítása!

8. FEJEZET:

Kapcsolódás a UNIX operációs rendszerhez

A UNIX operációs rendszer szolgáltatásai a C nyelvű programokból az ún. *rendszerhívásokon* keresztül érhetők el. Ezek a rendszerhívások lényegében adott feladatot ellátó függvények, amelyeket a felhasználói program hívhat. Ebben a fejezetben a C nyelvű programokból hívható legfontosabb rendszerhívásokat ismertetjük. Ha a UNIX operációs rendszer alatt dolgozunk, akkor ezek a függvények közvetlenül a segítségünkre lehetnek. A rendszerhívásokat gyakran alkalmazzuk a program maximális hatékonyságának elérése érdekében vagy a könyvtári függvényekkel nem megvalósítható feladatok ellátására. Abban az esetben, ha a C nyelvet nem UNIX operációs rendszerrel használjuk, akkor az itt közölt példákon keresztül betekinthetünk a C nyelvű programozás rejtel-meibe, és bár a részletek változhatnak, más operációs rendszer esetén is hasonló programok írhatók. Mivel az ANSI C könyvtár sok esetben a UNIX szolgáltatásait modellezi, az itt közölt programok a könyvtár jobb megismerését is segítik.

A fejezet anyaga három fő részre oszlik, az adatbevitel és adatkivitel, az állománykezelés és a tárkezelés műveleteire. Az első két rész feltételezi a UNIX jellemzőinek legalább alapfokú ismeretét.

A 7. fejezet az adatbevitel és adatkivitel olyan rendszerillesztési felületével foglalkozott, ami lényegében operációs rendszertől függetlenül egységes, mivel bármelyik konkrét rendszerben a standard könyvtár eljárásait a befogadó rendszer szolgáltatásainak figyelembevételével kell megírni. A következő néhány pontban a UNIX rendszer adatbevitellel és adatkivitellel kapcsolatos rendszerhívásait ismertetjük, és megmutatjuk, hogy ezekhez hogyan készíthetjük el a standard könyvtár megfelelő részeit.

8.1. Az állományleírók

A UNIX operációs rendszerben az összes adatbeviteli és adatkiviteli művelet állományok olvasásával vagy írásával valósul meg, mivel az összes perifériához való hozzáférés, beleértve a billentyűzetet és a képernyőt is, az állománykezelő rendszeren keresztül történik. Ez azt jelenti, hogy a felhasználói program és a perifériák közötti teljes adatcsere egyetlen homogén interfészen át bonyolódik le.

A legáltalánosabb esetben egy állomány olvasása vagy írása előtt szándékunkról tájékoztatni kell az operációs rendszert, és ezt a folyamatot az állomány *megnyitásának* nevezzük. Ha egy állományba írni akarunk, akkor szükség lehet az adott állomány létrehozására vagy a már meglévő állomány korábbi tartalmának törlésére. Az operációs rendszer ellenőrzi, hogy mindehhez van-e jogunk (A megadott állomány létezik-e? Van-e hozzáférési jogunk az állományhoz?), és ha mindent rendben talált, akkor visszatér a hívó programba egy kis, nem negatív egész számmal, amit *állományleírónak* nevezünk. Ezután minden esetben, amikor az állományból olvasni vagy abba írni akarunk, az állomány azonosítására az állománynév helyett ezt az állományleírót használjuk. (Az állományleíró a standard könyvtárban használt állománymutatóval vagy az MS-DOS-ban használt állománykezelővel analóg fogalom.) A megnyitott

állományra vonatkozó összes információt az operációs rendszer kezeli és a felhasználói program az állományra csak annak állományleírójával hivatkozik.

Mivel leggyakrabban a billentyűzeten és a képernyőn keresztüli adatbevitelt és adatkivitelt használjuk, ezért ezek kezelésére egy kényelmes megoldását fejlesztettek ki. Amikor az operációs rendszer parancsértelmezője (a shell) egy felhasználói programot futtat, ahhoz automatikusan három állományt nyit meg. Ezek (ahogyan erről már volt szó) a standard bemenet, a standard kimenet és a standard hibaállomány, amelyekhez rendre a 0, 1 és 2 állományleíró tartozik. Így ha egy program mindig a 0 leírójú állományt olvassa és az 1, ill. 2 leírójú állományba ír, akkor nem kell törődnie az állományok megnyitásával.

A felhasználó a < vagy > jelekkel átirányíthatja a program bemenetét vagy kimenetét a

```
prog <beallomany >kiallomany
```

formában. Ilyenkor a shell megváltoztatja a 0 és 1 állományleíróhoz tartozó alapértelmezés szerinti hozzárendelést az adott nevű állományokra. Normális esetben a 2 állományleíróhoz mindig a képernyő van hozzárendelve, így a hibaüzenetek mindig ott jelennek meg. Hasonló módon történik a bemenet és a kimenet kezelése pipeing mechanizmus alkalmazásakor. Minden esetben az állományok hozzárendelését az operációs rendszer (shell) változtatja meg és nem a felhasználói program. A programnak nincs tudomása arról, hogy honnan kapja a bemeneti adatokat és hová kerülnek a kimeneti adatok, mindössze csak azt tudja, hogy a 0 állomány bemenet, az 1 és 2 állomány kimenet.

8.2. Alacsony szintű adatbevitel és adatkivitel – a read és write függvények

A UNIX rendszerben az alacsony szintű adatbevitelt és adatkivitelt a `read` és `write` rendszerhívások intézik, amelyek a C nyelvű programból a `read` és `write` függvényekkel érhetők el. Mindkét függvény első argumentuma az állományleíró, a második argumentum pedig a felhasználói programban definiált karakteres tömb (puffer), amibe a bejövő adatok érkeznek, ill. aminek a tartalma kiíródik. A függvények harmadik argumentuma az átvinni kívánt bájtok (karakterek) száma. A függvények használatának módja:

```
int n_olvas = read(int fd, char *buf, int n);
int n_ir = write(int fd, char *buf, int n);
```

Mindegyik függvény a ténylegesen átvitt bájtok számával tér vissza, ami olvasáskor kisebb lehet, mint a híváskor megadott érték. A visszatérési érték `0`, ha állomány vége következett és `-1`, ha valamilyen hiba történt. Íráskor a visszatérési értéknek meg kell egyezni a kiíratni kívánt bájtszámmal, ha nem egyeznek, akkor hiba történt az átvitel közben.

Egy függvényhívással bármennyi bájt írható vagy olvasható. A leggyakoribb esetben a bájtszám 1, ami azt jelenti, hogy egy időben egyetlen karaktert írunk vagy olvasunk (puffereletlen adatátvitel). Gyakori még az 1024, 4096 vagy hasonló számú bájt átvitele, mivel ez a blokkméret jól illeszkedik a perifériák fizikai blokkméretéhez. Nagyobb méretű blokkok átvitele hatékonyabb, mivel fajlagosan kevesebb rendszerhívásra van szükség.

Az elmondottak alapján írjunk egy egyszerű programot, amely a bemenetről érkező adatokat átmásolja a kimenetre. (A program lényegében megegyezik az 1. fejezetben leírt másolóprogrammal.) Ez a program gyakorlatilag bármit bárhová átmásol, mivel a bemenet és a kimenet tetszőleges eszközre vagy állományba átirányítható.

```
#include "syscalls.h"

main() /* a bemenetet a kimenetre másolja */
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
        write (1, buf, n);
    return 0;
}
```

A rendszerhívások függvényprototípusait a `syscalls.h` headerben gyűjtöttük össze, ahonnan a fejezet programjaiba beiktathatók. Ez a header-név természetesen nem szabványban rögzített név.

A `BUFSIZ` paramétert szintén a `syscalls.h` headerben definiáltuk, értéke a helyi rendszerhez illeszkedően lett megválasztva. Ha az állomány mérete nem a `BUFSIZ` egész számú többszöröse, akkor a `read` a `write` függvénnyel kiírt bájtszámnál kisebb számmal fog visszatérni és a következő `read` visszatérési értéke nulla lesz (`EOF`).

Érdemes megnézni, hogy hogyan használható a `read` és a `write` magasabb szintű (`getchar` vagy `putchar` függvényekhez hasonló) függvények előállítására. Példa gyanánt írjuk meg a `getchar` pufferelletlen bemeneti függvényt, amely egy időben egy karaktert olvas a standard bemenetről.

```
#include "syscalls.h"

/* getchar: egykarakteres, pufferelletlen beolvasóeljárás */
int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char) c :
           EOF;
}
```

A programban `c` karakteres kell hogy legyen, mivel a `read` függvény karakteres mutatót igényel. A visszatéréskor `c`-re rákényszerített `unsigned char` típus garantáltan kizárja az előjel-kiterjesztésből adódó problémákat.

A `getchar` második változata egyszerre egy nagy adatblokkot olvas be és abból egyenként adja ki a karaktereket.

```
#include "syscalls.h"

/* getchar: egyszerű pufferelt változat */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) { /* a puffer üres */
```

```
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Ha a `getchar` ezen változatát az `<stdio.h>` header beiktatásával lefordítjuk, akkor a `getchar` nevet a `#undef` paranccsal definiálatlanná kell tenni, különben a rendszer a könyvtári makrót szerkesztené be a programba.

8.3. Az open, creat, close és unlink rendszerhívások

Az alapértelmezés szerinti standard bemenet, kimenet és hibaállomány kivételével az összes többi, írásra vagy olvasásra igénybe vett állományt explicit módon meg kell nyitni. Erre a célra két rendszerhívás, az `open` és a `creat` (vigyázat: nem create) használható.

Az `open` hasonló a 7. fejezetben leírt `fopen` függvényhez, kivéve, hogy állománymutató helyett egy állományleíróval tér vissza, ami `int` típusú.

Az `open` visszatérési értéke `-1`, ha a művelet közben valamilyen hiba történt. Az `open` használatát mutatja be a következő programrészlet.

```
#include <fcntl.h>

int fd;
int open(char *nev, int jelzo, int eng);

fd = open(nev, jelzo, eng);
```

Csakúgy, mint a `fopen` esetén, a `nev` argumentum az állomány nevét tartalmazó karaktersorozat. A második, `jelzo` argumentum `int` típusú és azt mondja meg, hogy az állományt milyen célból nyitottuk meg. Gyakran előforduló értékei:

`O_RDONLY`	megnyitás csak olvasásra;
`O_WRONLY`	megnyitás csak írásra;
`O_RDWR`	megnyitás írásra és olvasásra.

Ezek az állandók System V UNIX rendszer esetén az `<fcntl.h>` headerben, a Berkeley (BSD) változat esetén pedig a `<sys/file.h>` headerben vannak definiálva.

Egy létező állomány megnyitása olvasásra:

```
fd = open(nev, O_RDONLY, 0);
```

Az open harmadik, `eng` argumentuma ilyen típusú alkalmazások esetén mindig nulla. (Az `eng` paraméter használatára még visszatérünk.)

Ha egy nem létező állományt akarunk megnyitni, akkor hibajelzést kapunk. Egy új állomány létrehozása vagy egy meglévő állomány felülírása a `creat` rendszerhívással lehetséges. Ennek általános alakja:

```
int creat(char *nev, int eng);
fd = creat(nev, eng);
```

A `creat` függvény a hívása után az állományleíróval tér vissza, ha képes volt létrehozni a kívánt állományt, vagy a `-1` értékkel, ha nem. Ha a `creat`-nek megadott állomány már létezett, akkor a függvény a hosszúságát nullára állítja, amivel a korábbi tartalmat törli. Nem hiba a `creat` függvénynek már létező állomány nevét megadni.

Ha az állomány még nem létezik, akkor a `creat` az `eng` argumentumban megadott védelmi móddal hozza azt létre. A UNIX rendszerben minden állományhoz egy kilenc-bites védelmi kód tartozik, ami az állomány írási, olvasási, végrehajtás-hozzáférési, tulajdonosi és tulajdonoscsoporthoz tartozó engedélyeket tartalmazza. A védelmi kódot legkényelmesebben egy háromjegyű oktális számmal adhatjuk meg. Így pl. a `0755` kód a tulajdonosnak engedélyezi az írást, olvasást és a végrehajtást, a csoport többi tagjának és mindenki másnak is viszont csak olvasást és végrehajtást enged meg.

Az elmondottak illusztrálására ismertetjük a UNIX `cp` programjának egyszerűsített változatát, amely egy állományt egy másik állományba másol. Az itt közölt változat csak egy állományt másol, és nem engedi meg, hogy a második argumentum egy könyvtár legyen. Ugyancsak egyszerűsíti a feladatot, hogy a program rögzített védelmi kódot használ.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define ENG 0666 /* olvasás-írás a tulajdonosnak,
a csoportnak és másoknak */

void error (char *, ...);

/* cp: f1 másolása f2-be */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    if (argc != 3)
        error("Felhasználás: cp a-ból b-be");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: nem nyitható meg %s", argv[1]);
    if ((f2 = creat(argv[2], ENG)) == -1)
        error("cp: nem hozható létre %s, mód %03o",
            argv[2], ENG);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: írási hiba a(z) %s állományban", argv[2]);
    return 0;
}
```

A program `0666` védelmi kóddal egy kimeneti állományt hoz létre. A rögzített védelmi kód helyett felhasználhatjuk a bemeneti állomány eredeti védelmi kódját is, amit a 8.6. pontban ismertetendő `stat` rendszerhívással kérdezhetünk le.

Vegyük észre, hogy az `error` függvényt változó hosszúságú argumentumlistával hívjuk, hasonlóan a korábban ismertetett `printf` függvényhez. Az `error` függvény programja egyben példát mutat a `printf` függvénycsalád egy újabb tagjának használatára. A standard könyvtár `vprintf` függvénye a `printf` függvényhez hasonló, annyi eltérés-

sel, hogy a változó hosszúságú argumentumlistát egyetlen, a va_start makró hívásával inicializált argumentum helyettesíti. A haosnlóan kialakított vfprintf és vsprintf függvények az fprintf és sprintf függvényeknek felelnek meg.

```
#include <stdio.h>
#include <stdarg.h>

/* error: kiír egy hibaüzenetet és leállítja a program futását */
void error (char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "Hiba: ");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

A program által egy időben megnyitható állományok száma korlátozott, de ez a korlát gyakran 20 körül van. Ezért minden programot, amely több állományt használ, úgy kell kialakítani, hogy az állományleírók újra felhasználhatók legyenek. Az állományleíró és a megnyitott állomány közötti kapcsolat a close(int fd) függvénnyel szakítható meg, az így felszabaduló állományleíró más állományokhoz használható. A close függvény lényegében megfelel a standard könyvtár fclose függvényének, kivéve, hogy a puffert nem üríti ki. A programot a main függvényében kiadott exit vagy return utasítással leállítva az összes megnyitott állomány automatikusan lezáródik.

Az unlink(char *nev) függvény eltávolítja a nev nevű állományt az állománykezelő rendszerből. Az unlink a standard könyvtár remove függvényének felel meg.

8.1. gyakorlat. Írjuk újra a 7. fejezetben megismert cat programot úgy, hogy a standard könyvtári függvények helyett a read, write, open és close függvényeket használjuk! Végezzünk kísérleteket a két változat futási idejének meghatározása!

8.4. A véletlenszerű hozzáférés – az lseek függvény

Normális körülmények között a bemenet és a kimenet szekvenciális: minden egyes read vagy write hívással az állomány következő karakterpozíciójához férünk hozzá. Szükség esetén azonban az állomány tetszőleges sorrendben írható vagy olvasható. Ezt az lseek rendszerhívás teszi lehetővé, amellyel tényleges olvasás vagy írás nélkül tetszőlegesen mozoghatunk az állományban. Az lseek általános alakja:

```
long lseek(int fd, long offset, int bazis);
```

A függvény hívásakor az fd állományleíróval kijelölt állomány aktuális hozzáférési pozícióját az offset-nek megfelelő helyre állítja. Ez a hely egy relatív pozíció a bazis kezdőponthoz képest. Az állomány soron következő írása vagy olvasása az lseek függvénnyel beállított helyen fog kezdődni. A bazis kezdőpont értéke 0, 1 vagy 2 lehet attól függően, hogy az offset-et az állomány elejétől, az aktuális pozíciótól vagy az állomány végétől számoljuk. Például ha egy állományhoz további adatokat akarunk

hozzáfűzni (ez a UNIX shell >> átirányítási parancsával vagy az `fopen "a"` hozzáférési módjával valósítható meg), akkor az írás előtt meg kell keresni az állomány végét, amit az

```
lseek(fd, 0L, 2);
```

utasítással érhetünk el. Hasonló módon az állomány elejére pozicionálás („visszatekercselés", rewind) az

```
lseek(fd, 0L, 0);
```

utasítás hatására jön létre.

A `0L` értékű argumentumot `(long) 0` formában is írhatnánk, vagy megfelelően deklarált `lseek` esetén akár `0` formában.

Az `lseek` függvény felhasználásával az állományok a nagyméretű tömbökhöz hasonlóan kezelhetők, de az adatokhoz való hozzáférés nagyon lelassul. A következő példaprogram egy állomány tetszőleges helyéről tetszőleges számú bájtot olvas, és visszatér a beolvasott bájtok számával, vagy ha olvasás közben hiba történt, akkor `-1` értékkel.

```
#include "syscalls.h"

/* get: n db bájtot olvas a pos pozíciótól kezdve */
int get(int fd, long pos, char *buf, int n)
{
    if (lseek(fd, pos, 0) >= 0) /* beáll a pos helyre */
        return read(fd, buf, n);
    else
        return -1;
}
```

Az `lseek` visszatérési értéke `long` típusú és megadja az állományon belüli új pozíciót, vagy ha hiba fordult elő, akkor `-1` értékű. A standard könyvtár `fseek` függvénye hasonló az `lseek` függvényhez, annyi különbséggel, hogy az `fseek` első argumentuma `FILE *` típusú és a visszatérési értéke nem nulla, ha hiba jelentkezett.

8.5. Példa: az fopen és getc függvények megvalósítása

A következőkben az eddigieket megpróbáljuk egységbe foglalni az `fopen` és `getc` könyvtári függvények megvalósításával.

Emlékezzünk arra, hogy a standard könyvtári függvények az állományokat az állománymutatóval írják le és nem pedig az állományleíróval. Az állománymutató egy struktúrát címez, amelyben az állományra vonatkozó különböző információk (egy puffert címző mutató, amit felhasználva az állomány nagyobb blokkokban kezelhető; a pufferban maradt karakterek száma; a puffer következő karakterét címző mutató; az állományleíró; az írási–olvasási módot megadó jelzők; hibaállapotjelzők stb.) találhatók.

Az állományokat leíró adatstruktúra a `<stdio.h>` headerben van, amit minden olyan forrásállományba be kell építeni (`#include` utasítással), amely a standard bemenet-kimeneti könyvtár eljárásait használja. Természetesen a könyvtár függvényeit is be kell építeni a forrásprogramba. A következőkben ismertetjük az `<stdio.h>` egy részletét. Azok a nevek, amelyeket csak a könyvtár függvényei használhatnak, aláhúzással kezdődnek, ami csökkenti annak esélyét, hogy véletlenül megegyezzen egy, a programban használt névvel. Ezt a jelölésmódot használja az összes standard könyvtári eljárás.

```
#define   NULL     0
#define   EOF      (-1)
#define   BUFSIZ   1024
#define   OPEN_MAX 20 /* egy időben nyitott
          állományok száma */

typedef struct _iobuf {
   int cnt;
   /* a pufferban maradt karakterek száma */
   char *ptr;       /* a következő karakterpozíció */
   char *base;      /* a puffer kezdőcíme */
   int flag;        /* az állomány-hozzáférés módja */
   int fd;               /* az állományleíró */
} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin           (&_iob[0])
#define stdout          (&_iob[1])
#define stderr          (&_iob[2])

enum _flags {
   _READ  = 01,     /* állomány megnyitása olvasásra */
   _WRITE = 02,     /* állomány megnyitása írásra */
   _UNBUF = 04,     /* az állomány pufferelетlen */
   _EOF   = 010,    /* az állományban EOF található */
   _ERR   = 020     /* az állományban hiba volt */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)         (((p)->flag & _EOF) != 0)
#define ferror(p)       (((p)->flag & _ERR) != 0)
#define fileno(p)       ((p)->fd)

#define getc(p)         (--(p)->cnt >= 0 ? (unsigned \
        char) \
                        *(p)->ptr++ : _fillbuf(p))
#define putc(x, p)      (--(p)->cnt >= 0 ? *(p)->ptr++ \
                        = (x) : _flushbuf((x), p))
#define getchar()       getc(stdin)
#define putchar(x)      putc((x), stdout)
```

A `getc` makró normális esetben dekrementálja a darabszámot, lépteti a mutatót és visszatér a karakterrel. (Emlékeztetőül: a \ azt jelzi a fordítóprogramnak, hogy a definíció a következő sorban folytatódik!) Ha a pufferban maradt karakterek száma (vagyis a darabszám) negatív lesz, a `getc` hívja a `_fillbuf` függvényt, ami újra feltölti a puffert, inicializálja a struktúra tartalmát és visszatér egy karakterrel. A visszatéréskor adott karakter `unsigned` típusú, ami garantálja, hogy az összes karakter pozitív lesz.

Bár részleteiben nem tárgyaljuk, mégis beiktattuk a `putc` függvény definícióját is, annak bemutatására, hogy lényegében ugyanúgy működik, mint a `getc` függvény,

vagyis ha a puffer megtelt, hívja a `_flushbuf` függvényt. A közölt részlet a hibaállapotot, az állomány végét és az állományleírót kezelő makrókat is tartalmazza.

Ennyi bevezető információ birtokában már megírhatjuk az `fopen` függvényt! Az `fopen` legnagyobb része azzal foglalkozik, hogy megnyitja az állományt, a kívánt helyre pozicionál és a helyes állapotnak megfelelően állítja be a jelzőbiteket. Az `fopen` nem foglalja le a pufferterületet, ezt az első olvasáskor a `_fillbuff` teszi meg.

```
#include <fcntl.h>
#include "syscalls.h"
#define ENG 0666 /* írás, olvasás a tulajdonosnak,
a csoportnak és másoknak */

/* fopen: megnyit egy állományt, visszatér az állomány-
mutatóval */
FILE *fopen(char *nev, char *mod)
{
    int fd;
    FILE *fp;

    if (*mod != 'r' && *mod != 'w' && *mod != 'a')
        return NULL;
    for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
        if ((fp->flag & (_READ | _WRITE)) == 0)
            break; /* szabad területet talált */
    if (fp >= _iob + OPEN_MAX) /* nincs szabad hely */
        return NULL;

    if (*mod == 'w')
        fd = creat(nev, ENG);
    else if (*mod == 'a') {
        if ((fd = open(nev, O_WRONLY, 0)) == -1)
            fd = creat(nev, ENG);
        lseek(fd, 0L, 2);
    } else
        fd = open(nev, O_RDONLY, 0);
    if (fd == -1) /* a nev nem érhető el */
        return NULL;
    fp->fd = fd;
    fp->cnt = 0;
    fp->base = NULL;
    fp->flag = (*mod == 'r') ? _READ : _WRITE;
    return fp;
}
```

Az `fopen` itt ismertetett változata nem kezeli a szabványban megengedett összes hozzáférési módot, de ezek utólag viszonylag könnyen beépíthetők a programba. A program a "b" bináris hozzáférést sem kezeli, de ennek UNIX operációs rendszer esetén nincs is jelentősége. Ezenkívül nem veszi figyelembe az írásra és olvasásra egyaránt igénybe vehető állományt jelző "+" hozzáférési módot sem.

A `getc` adott állományra vonatkozó első hívásakor a darabszám nulla, ami a `_fillbuf` függvény hívását eredményezi. Ha a `_fillbuf` úgy találja, hogy az állo-

mány nincs megnyitva olvasásra, akkor azonnal EOF jelzéssel tér vissza. Megnyitott állomány esetén pedig megpróbál a puffer számára tárterületet lefoglalni (ha az olvasás pufferelt).

Ha a puffer létrejött (lefoglalta a területet számára), akkor a _fillbuf hívja a read függvényt, amely azt adatokkal tölti fel, beállítja a darabszámot és a mutatókat, majd a puffer elején lévő karakterrel tér vissza. A _fillbuf a további hívásoknál már a meglévő puffert használja.

```
#include "syscalls.h"

/* _fillbuf: területet foglal a puffernek és feltölti */
int _fillbuf(FILE *fp)
{
    int bufsize;

    if ((fp->flag&(_READ|_EOF|_ERR)) != _READ)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL) /* még nincs puffer */
        if ((fp->base = (char *) malloc(bufsize)) == NULL)
            return EOF; /* nincs hely a puffer számára */
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->flag |= _EOF;
        else
            fp->flag |= _ERR;
        fp->cnt = 0;
        return EOF;
    }
    return (unsigned char) *fp->ptr++;
}
```

Most már csak az a kérdés, hogy hogyan indul az egész folyamat? Az stdin, stdout és stderr számára definiálni és inicializálni kell az _iob tömböt:

```
FILE _iob[OPEN_MAX] = { /* stdin, stdout, stderr: */
    { 0, (char *) 0, (char *) 0, _READ, 0 },
    { 0, (char *) 0, (char *) 0, _WRITE, 1 },
    { 0, (char *) 0, (char *) 0, _WRITE | _UNBUF, 2 }
};
```

A struktúra flag részének inicializálása mutatja, hogy stdin olvasható, stdout írható és stderr írható, pufferelletlen hozzáférésű.

8.2. gyakorlat. Írjuk át az fopen és _fillbuf függvényeket úgy, hogy az explicit bitműveletek helyett bitmezőket használunk! Hasonlítsuk össze a két változat forrásprogramjának méretét és a futási időket!

8.3. gyakorlat. Tervezzük meg és írjuk meg a _flushbuf, fflush és fclose függvényeket!

8.4. gyakorlat. A standard könyvtár

```
int fseek(FILE *fp, long offset, int bazis)
```

függvénye megegyezik az `lseek` függvénnyel, kivéve, hogy az `fp` állománymutatót használja az állományleíró helyett és hogy a visszatérési értéke az `int` típusú állapotjelzés, nem pedig egy pozíció. Írjuk meg az `fseek` függvényt! Gondoskodjunk arról, hogy az általunk írt `fseek` pufferkezelése összhangban legyen a könyvtár többi függvényével!

8.6. Példa: katalógusok kiíratása

Néha az állománykezelő rendszerel az eddigiektől eltérő párbeszédet kell folytatnunk, pl. ha magának az állománynak a jellemzőire vagyunk kíváncsiak és nem pedig a tartalmára. Erre jó példa a katalóguslistázó program, ami feladatát tekintve megfelel a UNIX `ls` parancsának. A program kiírja a katalógusban lévő állományok nevét és opcionálisan még több más információt (méret, hozzáférési kód stb.) is. A parancs analóg az MS-DOS `dir` parancsával.

Mivel a UNIX katalógusa maga is egy állomány, az `ls` parancsnak csak be kell olvasni ezt az állományt és kikeresni belőle az állományok neveit, ill. ha szükséges, akkor egy rendszerhívással már meghatározható az állomány többi jellemzője is, mint pl. a mérete. Más operációs rendszerek (pl. MS-DOS) esetén az állományok nevéhez való hozzáférés is egy rendszerhívást igényel. Mi a programunkkal viszonylag rendszertől függetlenül akarunk az információkhoz hozzáférni, bár maga a megvalósítás nagymértékben függ a rendszertől.

Az elmondottakat az `fsize` program megírásával fogjuk illusztrálni. Az `fsize` program az `ls` parancs egy speciális változata, amely kiírja a parancssor-argumentumok listájában megadott állománynevekhez tartozó méretet. Ha az állományok egyike egy (al-)katalógus, akkor az `fsize` programot rekurzívan alkalmazzuk a katalógusra. Ha a programnak nincs argumentuma, akkor az aktuális katalógust dolgozza fel.

A feladat megoldását kezdjük a UNIX állománykezelő rendszerének leírásával! A katalógus egy állomány, amely az állományok neveinek listáját és az állományok helyére utaló információkat tartalmazza. A „hely" valójában egy másik táblázatba, az ún. inode táblázatba mutató index. Az *inode* táblázat adott állományhoz tartozó bejegyzése az állomány nevén kívül annak összes többi jellemzőjét tartalmazza. Egy katalógusbejegyzés csak két adatból, az állomány nevéből és egy inode számból áll.

Sajnos, egy katalógus konkrét formátuma és pontos tartalma az operációs rendszer egyes változatainál más és más, ezért a feladatot két részre bontjuk, amivel megpróbáljuk leválasztani a nem hordozható (rendszertől függő) elemeket. A program külső szintjén definiálunk egy struktúrát, amit `Dirent`-nek nevezünk és az `opendir`, `readdir`, ill. `closedir` eljárásokkal rendszertől függő módon férünk hozzá a katalógusbejegyzésben lévő névhez és inode számhoz. Ezeket az eljárásokat és a `Dirent` struktúrát használjuk szoftver-interfészként az `fsize` megírásánál. A rendszertől független részek megírása után megmutatjuk, hogy a rendszertől függő részek hogyan valósíthatók meg a UNIX Version 7 és System V változatánál használt katalógussal. A további változatokhoz tartozó megoldásokat meghagyjuk gyakorlatnak.

A `Dirent` struktúra az állomány nevét és inode számát tartalmazza. Az állománynév max. hosszát a rendszertől függő `NAME_MAX` érték határozza meg. Az `opendir` függvény egy `DIR` nevű struktúrát címző mutatóval tér vissza, amelyet a `readdir` és `closedir` függvények használnak. (A `DIR` struktúra analóg a `FILE` struktúrával.) Ezek a definíciók és adatok a `dirent.h` headerben vannak összegyűjtve.

```
#define NAME_MAX 14 /* a leghosszabb állománynév-
komponens, az érték a rendszertől függ */

typedef struct { /* a hordozható katalógusbejegyzés */
   long ino; /* inode szám */
   char name[NAME_MAX+1]; /* a név és a '\0' végjel */
} Dirent;

typedef struct { /* a minimális DIR: nincs pufferelés */
   int fd; /* a katalógus állományleírója */
   Dirent d; /* a katalógusbejegyzés */
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

A `stat` rendszerhívás veszi az állomány nevét, és az inode-ban található összes információt adja vissza, vagy `-1` értéket, ha hibát érzékelt. A

```
char *nev;
struct stat stbuf;
int stat(char *, struct stat *);

stat(nev, &stbuf);
```

programrészlet feltölti az `stbuf` struktúrát a `nev` nevű állomány inode-jában szereplő információval. A `stat` függvény által visszaadott struktúra leírása a `<sys/stat.h>` headerben van és tipikusan a következő módon néz ki:

```
struct stat /* a stat által visszaadott inode információk */
{
   dev_t  st_dev;  /* az inode eszköze (perifériája) */
   ino_t  st_ino;  /* az inode száma */
   short  st_mode; /* mód-bitek */
   short  st_nlink;/* az állományhoz tartozó linkek
                      száma */
   short  st_uid;  /* a tulajdonos azonosítója */
   short  st_gid;
   /* a tulajdonosi csoport azonosítója */
   dev_t  st_rdev; /* speciális állományok adata */
   off_t  st_size; /* az állomány mérete karakterben */
   time_t st_atime;/* az utolsó hozzáférés időpontja */
   time_t st_mtime;/* az utolsó módosítás időpontja */
   time_t st_ctime;/* az inode utolsó változtatásának
                      időpontja */
};
```

A felsorolt adatok többségét a megjegyzésben megmagyaráztuk. Az olyan típusok, mint a `dev_t` vagy az `ino_t` a `<sys/types.h>` headerben vannak definiálva, így a forrásprogramhoz azt is hozzá kell szerkeszteni egy `#include` utasítással.

Az st_mode bejegyzés az állományt leíró jelzőbiteket tartalmazza. A jelzőbitek definíciója szintén a <sys/stat.h> headerben található, itt csak az állomány típusát megadó jelzőkkel foglalkozunk:

```
#define S_IFMT 0160000 /* az állomány típusa */
#define S_IFDIR 0040000 /* katalógus */
#define S_IFCHR 0020000 /* speciális karakteres */
#define S_IFBLK 0060000 /* speciális blokkos */
#define S_IFREG 0100000 /* szabályos */

/* ... */
```

Ezek után már megírhatjuk az fsize programot! Ha a stat függvénytől kapott mód azt jelzi, hogy az állomány nem katalógus, akkor a mérete már a rendelkezésünkre áll és közvetlenül kiíratható. Ha az állomány egy katalógus, akkor azt állományonként fel kell dolgoznunk. Mivel egy katalógus további alkatalógusokat is tartalmazhat, a feldolgozás rekurzív lesz.

A main eljárás főleg a parancssor-argumentumokkal foglalkozik, előállítva az fsize argumentumait.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h> /* jelzők az olvasáshoz és íráshoz */
#include <sys/types.h> /* typedef utasítások */
#include <sys/stat.h> /* stat-ból visszaadott struktúra */
#include "dirent.h"

void fsize(char *);

/* az állományok méretének kiírása */
main(int argc, char **argv)
{
    if (argc == 1) /* alapfeltételezés szerint az
            az aktuális katalógus */
        fsize(".");
    else
        while (--argc > 0)
            fsize(*++argv);
    return 0;
}
```

Az fsize függvény kiírja az állomány méretét. Ha az állomány katalógus, akkor az fsize hívja a dirwalk függvényt, ami feldolgozza a katalógus összes állományát. Annak eldöntésére, hogy egy állomány katalógus-e vagy sem, a <sys/stat.h> headerben definiált S_IFMT és S_IFDIR jelzőbitek használhatók. A megfelelő programrészben ügyeljünk a zárójelezésre, mert az & precedenciája alacsonyabb, mint az == precedenciája.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn)(char *));
```

```
/* fsize: kiírja a nev nevű állomány méretét */
void fsize(char *nev)
{
    struct stat stbuf;

    if (stat(nev, &stbuf) == -1) {
        fprintf(stderr, "fsize: nem hozzáférhető %s\n", nev);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(nev, fsize);
    printf("%8ld %s\n", stbuf.st_size, nev):
}
```

A `dirwalk` függvény egy olyan általános eljárás, ami az argumentumában megadott függvényt használja fel egy adott katalógusban lévő állományokra. A `dirwalk` megnyitja a katalógust, ciklusban végigmegy az összes állományon, mindegyikre meghívja az átadott függvényt, majd lezárja a katalógust és visszatér. Mivel az `fsize` is hívja az egyes katalógusok esetén a `dirwalk` függvényt, a két függvény rekurzívan hívja egymást.

```
#define MAX_PATH 1024

/* dirwalk: fcn-t alkalmazza a dir katalógus összes állományára*/
void dirwalk(char *dir, void (*fcn)(char *))
{
    char nev[MAX_PATH];
    Dirent *dp;
    DIR *dfd;

        if ((dfd = opendir(dir)) == NULL) {
            fprintf(stderr, "dirwalk: nem nyitható meg
                    %s\n", dir);
            return;
        }
        while ((dp = readdir(dfd)) != NULL) {
            if (strcmp(dp->nev, ".") == 0 ||
                    strcmp(dp->, "..") == 0)
            continue; /* átugorja önmagát és a szülőt */
        if (strlen(dir)+strlen(dp->nev)+2 > sizeof(nev))
            fprintf(stderr, "dirwalk: a név %s/%s tul
                    hosszu\n", dir, dp->nev);
        else {
            sprintf(nev, "%s/%s", dir, dp->nev);
            (*fcn)(nev);
        }
    }
    closedir(dfd);
}
```

A `readdir` függvény a hívása után a következő állományt leíró információk mutatójával tér vissza, vagy `NULL` értékű mutatóval, ha nincs több állomány. Mindegyik kataló-

gus tartalmaz bejegyzést saját magáról (ennek a neve "."), valamint a szülőjéről (ennek a neve "..") és ezeket a programnak át kell lépni, különben rendkívüli mértékben megnőne a futási idő.

Az eddigi programok függetlenek voltak a katalógusok fizikai szerkezetétől. A következőkben bemutatjuk a rendszerfüggő `opendir`, `readdir` és `closedir` függvények egyszerűsített változatát. Ezek a programok a UNIX rendszer Version 7 vagy System V változatához használhatók és a katalógusokra vonatkozó információkat a `<sys/dir.h>` headerből veszik. A katalógusokat leíró információk fontosabb része:

```
#ifndef DIRSIZ
#define DIRSIZ 14 /* az állománynév hossza */
#endif
struct direct /* katalógusbejegyzés*/
{
    ino_t d_ino; /* inode szám */
    char d_name[DIRSIZ]; /* hosszú állománynév, */
};                       /* '\0' végjel nélkül */
```

Az operációs rendszer néhány változata hosszabb állományneveket is megenged és sokkal bonyolultabb szerkezetű katalógust használ.

Az `ino_t` típus `typedef` utasítással lett definiálva és az inode táblázat indexét írja le. Ez az operációs rendszer tulajdonságai alapján `unsigned short` típusú adat lehet, ami szabályosan használható is, de nem célszerű a programban rögzíteni, mert más operációs rendszer esetén más lehet. Így jobb megoldásnak tűnik a `typedef` utasítással beállított típus. A rendszertől függő adattípusok teljes halmaza a `<sys/types.h>` headerben található.

Az `opendir` függvény megnyitja a katalógusállományt, ellenőrzi, hogy annak tartalma tényleg katalógus-e (ezt az `fstat` rendszerhívással teszi, ami lényegében megegyezik a `stat` rendszerhívással, kivéve, hogy az állományleírót használja az állomány azonosítására), lefoglalja a tárban a katalógus adatait tároló struktúra helyét, majd beleolvassa az információt. Az `opendir` függvény:

```
int fstat(int fd, struct stat *);

/* opendir: megnyitja a katalógust a readdir hívása előtt */
DIR *opendir (char *dirname)
{
    int fd;
    struct stat stbuf;
    DIR *dp;

    if ((fd = open(dirname, O_RDONLY, 0)) == -1
    || fstat(fd, &stbuf) == -1
    || (stbuf.st_mode & S_IFMT) != S_IFDIR
    || (dp = (DIR *) malloc (sizeof (DIR))) == NULL)
        return NULL;
    dp->fd = fd;
    return dp;
}
```

A `closedir` függvény lezárja a katalógusállományt és felszabadítja a tárban lefoglalt helyet.

```
/* closedir: lezárja az opendir-rel megnyitott katalógust */
void closedir (DIR *dp)
{
    if (dp) {
        close (dp->fd);
        free(dp);
    }
}
```

A `readdir` függvény az egyes katalógusbejegyzések beolvasására a `read` függvényt használja. Ha egy katalógusbejegyzés aktuálisan nem használt (pl. mert az állományt töröltük), akkor az inode száma nulla és a `readdir` az ilyen bejegyzést átlépi. Máskülönben az inode számot és az állomány nevét elhelyezi egy `static` tárolási osztályú struktúrában, majd visszatér a sturktúra mutatójával. Minden `readdir` hívás felülírja az előző olvasáskor kapott információt.

```
#include <sys/dir.h> /* az adott rendszer katalógusának
szerkezete itt van leírva */

/* readdir: sorban beolvassa a katalógusbejegyzéseket */
Dirent *readdir(DIR *dp)
{
    struct direct dirbuf; /* a konkrét katalógusszerkezet */
    static Dirent d; /* visszatérés: hordozható szerkezet */
    while (read (dp->fd, (char *) &dirbuf, sizeof
            (dirbuf))== sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0) /* a bejegyzés helye */
            continue; /* nem használt */
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name [DIRSIZ] = '\0'; /* lezárja a karak- */
        return &d;              /* tersorozatot */
    }
    return NULL;
}
```

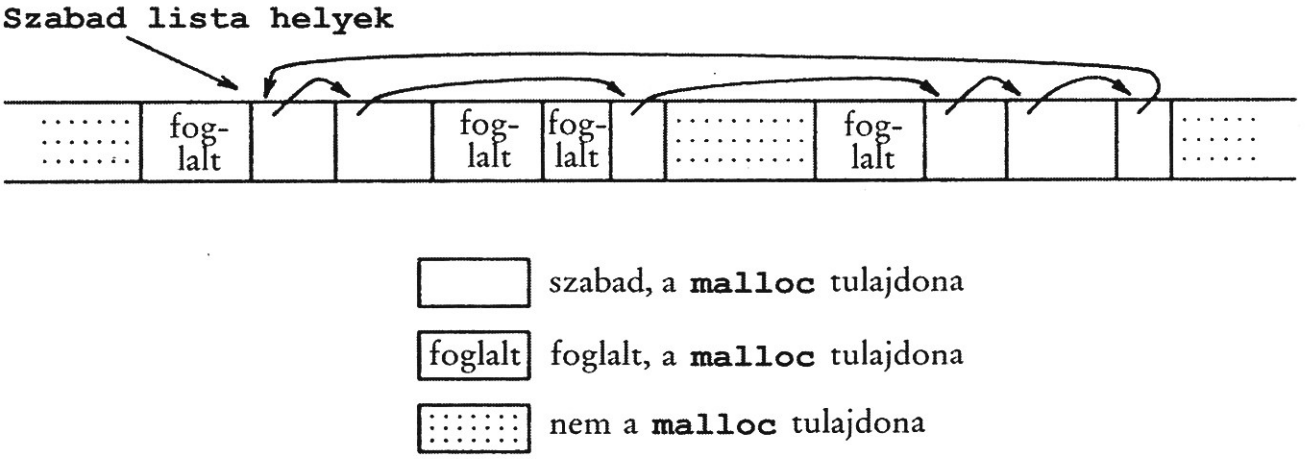
Bár az `fsize` program meglehetősen speciális, mégis számos fontos dolgot jól szemléltet. Az első fontos megjegyzés, hogy az `fsize` nem „rendszerprogram", csak olyan információt használ, amelynek formáját és tartalmát az operációs rendszer határozza meg. A második lényeges dolog, hogy ilyen programok esetén az információ rendszerfüggő leírása csak a standard headerben jelenjen meg és a program ezeket a header állományokat építse be, ahelyett, hogy saját maga deklarálná a géptől és rendszertől függő adatokat. További fontos programozási szempont, hogy a rendszerfüggő részekhez a lehető legnagyobb gonddal kell megtervezni az interfészeket, hogy a program többi része viszonylag rendszertől független lehessen. Erre a legjobb példát a standard könyvtár függvényeinél láthatunk.

8.5. gyakorlat. Módosítsuk az `fsize` programot úgy, hogy más, az inode táblázatban szereplő információt is kiírjon!

8.7. Példa: tárterület-lefoglaló program

Az 5. fejezetben bemutattunk egy korlátozott módon használható, veremorientált tárterület-foglaló programot. A most megírt változat nem tartalmaz korlátozásokat, a `malloc` és a `free` hívásai tetszőleges sorrendben történhetnek és a `malloc` szükség esetén az operációs rendszertől további tárterületet igényelhet. A tárterület-foglaló program itt megírt eljárásai jól példázzák, hogy hogyan lehet géptől függő programot viszonylag gépfüggetlen módon megírni. A programban megmutatjuk a struktúrák, unionok és a `typedef` utasítás gyakorlati alkalmazását is.

A `malloc` program szükség esetén az operációs rendszertől igényel tárterületet, szemben az 5. fejezetben leírt programmal, amely a fordításkor rögzített méretű tömb elemeivel gazdálkodott. Mivel a program más tevékenységei a `malloc` hívása nélkül is igényelhetnek tárterületet, ezért a `malloc` eljárással kezelt tárterület nem összefüggő. Emiatt a szabad tárolóhelyeket a szabad blokkok listájaként tartjuk nyilván. Minden blokk tartalmazza a méretét, a következő blokk mutatóját, valamint magát a tárterületet. A listában a blokkok növekvő tárcímek szerint rendezettek és az utolsó (legnagyobb című) blokk a legelső blokkra mutat. A viszonyokat jól szemlélteti a következő ábra.



Ha igény érkezik, akkor a program végignézi a szabad blokkok listáját és az első elegendően nagy blokkot adja vissza. Ezt az algoritmust a „legelső illeszkedés" algoritmusnak nevezzük, szemben a „legjobb illeszkedés" algoritmussal, amely az igényt még kielégítő legkisebb blokkot adja vissza. Ha a blokk mérete pontosan megegyezik az igényelt mérettel, akkor kiemeljük a szabad blokkok listájából és átadjuk a felhasználónak. Ha a talált szabad blokk túl nagy, akkor a program leválasztja belőle a kívánt részt és átadja a felhasználónak, a maradékot pedig meghagyja a szabad blokkok listájában (természetesen módosítva a jellemzőit). Ha a listában nincs elegendően nagy blokk, akkor a program az operációs rendszertől egy nagyobb tárterületet kér és hozzácsatolja a szabad blokkok listájához.

A tárterület felszabadításakor szintén végig kell nézni a szabad blokkok listáját és megkeresni azt a helyet, ahová (a címe alapján) a felszabadult blokk beilleszthető. Ha a felszabadult blokk egyik oldalával illeszkedik egy szabad blokkhoz, akkor a program ezeket egybeolvasztja egyetlen nagyobb blokká, így a tárterület nem forgácsolódik szét kis részekre. A szomszédos helyzet meghatározása a címek szerinti rendezettség miatt egyszerű.

Az egyik fő probléma, amivel már az 5. fejezetben is foglalkoztunk, hogy a `malloc` által visszaadott tárterületnek meghatározott illesztési feltételeket kell kielégíteni ahhoz, hogy az objektumainkat ezen a területen tárolni tudjuk. Bár a számítógépek társzervezése nagymértékben különbözhet, minden gép esetén létezik egy olyan alap-

vető adattípus, amely ha tárolható az adott címen, akkor minden más adattípus is tárolható ott. Néhány számítógép esetén ez az alapvető adattípus a `double`, más gépeknél viszont az `int` vagy a `long`.

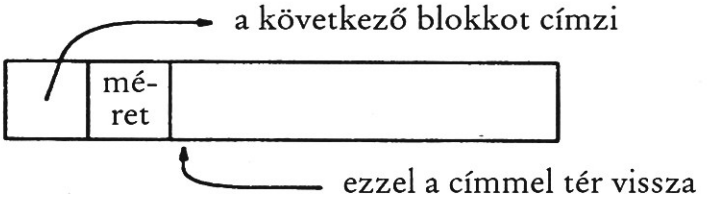
Egy szabad blokk tartalmazza a láncban utána következő blokk mutatóját, valamint a blokk méretét és ezután következik maga a szabad tárterület. A blokk elején lévő vezérlő információt fejnek nevezzük. A tárillesztés egyszerűsítése érdekében minden blokk mérete a fej méretének egész számú többszöröse és a fej pedig megfelelően illeszkedik. Ezt az adatszerkezetet egy unionnal érhetjük el, amely tartalmazza a fej struktúráját és kielégíti az illesztés szempontjából alapvető adattípusra vonatkozó igényeket. Ezt az alapvető adattípust a program `long`-nak tekinti. Az így kialakított adatszerkezet:

```
typedef long Align; /* illesztés long határhoz */

union header { /* a blokk feje */
    struct {
        union header *ptr; /* a következő blokk címe */
        unsigned size; /* a blokk mérete */
    } s;
    Align x; /* a blokk kényszerített illesztése */
}
typedef union header Header;
```

Az `Align` mezőt soha nem használjuk, csak azzal kényszerítjük a fejet az illesztési feltételek kielégítésére.

A `malloc` a karakterben igényelt méretet felkerekíti a fejméret egész számú többszörösére. A ténylegesen kiutalt blokk mérete ennél eggyel nagyobb (egy egységnyi hely kell magának a fejnek is) és ezt a méretet írja a program a fej `size` változójába. A `malloc` által visszaadott mutató a blokk szabad területének kezdetére és nem a fejre mutat. A felhasználó a kapott tárterülettel bármit csinálhat, de ha a kiutalt területen kívülre ír, akkor valószínűleg adatvesztés és ebből adódó hiba jön létre. A blokk méretét megadó mezőre szükség van, mivel a `malloc` által kezelt blokkok nem összefüggő, folytonos sorozatot alkotnak, így a méretük nem számítható ki a címaritmetikával.



A `base` változót használjuk a folyamat indulásakor. Ha a `freep` (ami a szabad blokkok listájának kezdetét kijelölő mutató) értéke `NULL`, ami a `malloc` első hívásakor biztosan igaz, akkor egy elfajult szabad lista alakult ki: ez egyetlen nulla méretű blokkot tartalmaz, amely saját magára mutat. A program minden esetben végigkeresi a szabad listát és a megfelelő méretű szabad blokk keresését a `freep` blokknál kezdi (ami az utoljára talált szabad blokk helye). Azzal, hogy az üres blokkok listáját nem mindig az első (legkisebb című) blokkal kezdjük végignézni, a lista hosszabb használat után is homogén marad. Ha a program egy túl nagy blokkot talál, akkor annak a végéből levágott megfelelő területtel tér vissza a felhasználóhoz. Ezzel a módszerrel a blokk eredeti fejében csak a méretet kell módosítani. A felhasználónak visszaadott mutató a blokk első szabad helyét címzi (ami közvetlenül a fej utáni első hely). A program:

```
static Header base; /* üres lista az induláshoz */
static Header *freep = NULL; /* az üres lista kezdete */

/* malloc: általános célú tárterület-foglaló program */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) { /* nincs még
            szabad lista */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) { /* elég nagy a hely */
            if (p->s.size == nunits) /*a méretek egyeznek */
                prevp->s.ptr = p->s.ptr;
            else { /* kiadja a blokk végét */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep) /* körbement a listán */
            if ((p = morecore(nunits)) == NULL)
                return NULL; /* nincs több hely */
    }
}
```

A `morecore` függvény az operációs rendszertől igényel további területet. Az, hogy ezt hogyan csinálja, az alkalmazott operációs rendszertől függ. A tárterület operációs rendszertől való kérése viszonylag „költséges" (főleg időigényes) művelet, ezért ezt nem akarjuk minden `malloc` híváskor megtenni és a `morecore` függvénnyel szükség esetén egy nagyobb, legalább `NALLOC` egységből álló területet kérünk. A terület méretének beállítása után a `morecore` függvény a `free` függvényt felhasználva iktatja be ezt a nagyobb területet a szabad blokkok listájába.

A UNIX `sbrk(n)` rendszerhívása egy `n` bájtos tárterületet címző mutatóval tér vissza. Ha nincs tárterület, akkor a `sbrk` visszatérési értéke `-1` (bár jobb lett volna, ha a visszatérési érték `NULL`). A `-1` értéket a `char *` kényszerített típuskonverzióval kell átalakítani, hogy az összehasonlítható legyen a függvény visszatérési értékével. A kényszerített típuskonverziók miatt a függvény viszonylag érzéketlen a különböző számítógépek mutatóábrázolásával szemben. Van még egy feltétel, amit a `morecore` függvénynek ki kell elégíteni: a különböző blokkok `sbrk` függvény által visszaadott mutatóinak összehasonlíthatóaknak kell lenni. Ez a szabvány szerint nem garantálható, mert az csak az azonos tömbhöz tartozó két mutató összehasonlíthatóságát írja elő. Így a `malloc` függvény itt közölt változata nem teljesen hordozható, csak olyan

rendszerek esetén használható, amelyek lehetővé teszik a mutatók általános összehasonlítását. Az elmondottak alapján kialakított `malloc` függvény:

```
#define NALLOC 1024 /* a minimális terület */
/* a malloc által használt egységekben */

/* morecore: az operációs rendszertől tárterületet kér */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1) /* nincs több terület */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *)(up+1));
    return freep;
}
```

A tárkezelő programok közül a `free` maradt utoljára. A függvény végignézi a szabad blokkok listáját és a megfelelő helyre beiktatja a felszabadult blokkot. A beiktatás két üres blokk közé vagy a lista végére történhet. Bármelyik esetben, ha a felszabadult blokk szomszédos egy szabad blokkal, akkor a `free` a két blokkot összevonja. Itt csak arra kell ügyelni, hogy a mutató a megfelelő helyet címezze és a méret helyes legyen.

```
/* free: visszarak egy blokkot a szabad blokkok listájába */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *)ap - 1; /* a blokk fejére mutat */
    for (p = freep; !(bp > p && bp < p->s.ptr);
         p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break; /* a felszabadult blokk a lista elejére
            vagy végére kerül */
    if (bp + bp->s.size == p->s.ptr) {
    /* a felső szomszédhoz kapcsoljuk */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {
    /* az alsó szomszédhoz kapcsoljuk */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
```

```
        p->s.ptr = bp;
    freep = p;
}
```

Bár a tárolókezelési műveletek alapvetően gépfüggőek, a programok jól mutatják, hogy a gépfüggés lekezelhető és a program viszonylag kis részére korlátozható. A `typedef` utasítás és az union felhasználásával a tárilleszkedési feltételek kielégíthetők (feltéve, hogy az `sbrk` függvény a megfelelő mutatót adja). A kényszerített típusmódosítás explicitté teszi a mutató konverzióját és még a rosszul tervezett rendszercsatlakozás okozta problémát is megoldja. A programban leírt részletek a tárkezelésre vonatkoznak, de az elvek és a megközelítés más esetben is jól használható.

8.6. gyakorlat. A standard könyvtárban található `calloc(n, size)` függvény `n` darab `size` méretű objektum számára lefoglalt és nulla kezdeti értékkel feltöltött tárolóterület mutatójával tér vissza. Írjuk meg a `calloc` függvényt úgy, hogy az hívja a `malloc`-ot, vagy megfelelően módosítsuk a `malloc` függvényt!

8.7. gyakorlat. A `malloc` a kért méretet ellenőrzés nélkül elfogadja és a `free` feltételezi, hogy a felszabadítandó blokk mérete érvényes. Javítsuk ki úgy ezeket a programokat, hogy nagyobb gondot fordítsanak a hibaellenőrzésre!

8.8. gyakorlat. A `malloc` és a `free` függvények felhasználásával írjuk meg a `bfree(p, n)` függvényt úgy, hogy az felszabadítsa az `n` karakterből álló tetszőleges `p` blokkot. Ezt a `bfree` függvényt alkalmazva a felhasználó bármikor beiktathat a szabad blokkok listájába egy statikus vagy külső tömböt.

Referencia-kézikönyv

A1. Bevezetés

Ez a kézikönyv a C nyelv ANSI felé 1988. október 31-én benyújtott és az „Amerikai Nemzeti Szabvány Információs Rendszerekre – A C Programozási Nyelv, X3.159-1989." címmel elfogadott szabvány alapján készült. A kézikönyv a tervezett szabvány értelmezése és nem magának a szabványnak a leírása, bár gondosan ügyeltünk arra, hogy a nyelv megbízható leírását adjuk.

Ez a leírás a legtöbb részletében követi a szabvány felépítését (ami a könyv első kiadása után jelent meg), de szerkezete a részletekben különbözik attól. Néhány fogalom átnevezésétől, a lexikális tokenek (szintaktikai egységek) nem formalizált leírásától vagy az előfeldolgozó rendszertől eltekintve az itt leírt szintaxis megfelel a szabványban foglaltaknak.

> Bár ez a rész egy kézikönyv, az egyes pontokhoz magyarázatokat fűztünk, amit kisebb betűtípussal szedtünk. A magyarázatok többsége rávilágít arra, hogy az ANSI C miben különbözik a könyv első kiadásában definiált nyelvtől, vagy hogy a különböző fordítóprogramok esetén milyen finomítások érvényesek.

A2. Lexikális megállapodások

Egy program egy vagy több fordítási egységből áll, amelyek állományokban helyezkednek el. A program feldolgozása több fázisban történik, az egyes fázisokat az A12. pontban foglaltuk össze. A legelső feldolgozási fázisban a program elvégzi az alacsony szintű lexikális átalakítást, amelynek során először végrehajtódnak a # jellel kezdődő sorokban elhelyezett direktívák, majd megtörténik a makródefiníciók feldolgozása és végül létrejön a makrókifejtés. Az A12. pontban leírt előfeldolgozás befejeztével a program szintaktikai egységek (tokenek) sorozatára egyszerűsödik.

A2.1. Szintaktikai egységek

A szintaktikai egységek hat osztályba sorolhatók: azonosítók, kulcsszavak, állandók, karaktersorozatok, operátorok és egyéb szeparátorok. A szóközt, a vízszintes és függőleges tabulátort, az új sort, a lapemelést és a megjegyzéseket (közös néven üres helyeket) a fordítóprogram nem veszi figyelembe, kivéve ha szintaktikai egységeket választanak el egymástól. Valamennyi üres helyre szükség van a szomszédos azonosítók, kulcsszavak és állandók elválasztásához.

Ha a beolvasott programszöveg adott karakterig szintaktikai egységekre lett bontva, akkor a fordítóprogram a következő szintaktikai egységnek azt a leghosszabb karaktersorozatot tekinti, amelyről feltételezhető, hogy egyetlen szintaktikai egységet alkot.

A2.2. Megjegyzések

A megjegyzés szövege a /* karakterekkel kezdődik és a */ karakterekkel zárul. A megjegyzések nem ágyazhatók egymásba és nem fordulhatnak elő karaktersorozatokban vagy karakteres állandókban.

A2.3. Azonosítók

Egy azonosító betűkből és számjegyekből áll. Az első karakterének betűnek kell lenni és az _ aláhúzás-karakter betűnek számít. Azonosítókban a nagy- és kisbetűk különböznek. Az azonosítók hossza tetszőleges lehet és belső azonosítók esetén legalább 31 karakter szignifikáns, de néhány rendszerben a szignifikáns karakterek száma több is lehet. Belső azonosítók közé tartozik az előfeldolgozó rendszerrel értelmezett makrónév és minden más név, amelynek nincs külső csatolása (l. az A11.2. pontot). A külső csatolású azonosítókra ennél több megszorítás érvényes: a megvalósításokban csak az első hat karakter szignifikáns és nem tesznek különbséget kis-, ill. nagybetű között.

A2.4. Kulcsszavak

A következő azonosítók fenntartott kulcsszavak és más célra nem használhatók:

```
auto      double    int       struct
break     else      long      switch
case      enum      register  typedef
char      extern    return    union
const     float     short     unsigned
continue  for       signed    void
default   goto      sizeof    volatile
do        if        static    while
```

Néhány megvalósításban fenntartott szó még az `asm` és a `fortran`.

> A `const`, `signed` és `volatile` új az ANSI szabványban; az `enum` és a `void` a könyv első kiadásához képest új, de már korábban is általánosan használt kulcsszó. Az `entry` régebben fenntartott, de soha nem használt kulcsszó volt, ezért a továbbiakban már nem fenntartott kulcsszó.

A2.5. Állandók

A C nyelvben többféle állandó létezik, ezek mindegyikéhez egy adattípus tartozik. Az alapvető adattípusok leírása az A4.2. pontban található.

állandók:
- *egész_állandó*
- *karakteres_állandó*
- *lebegőpontos_állandó*
- *felsorolt_állandó*

A2.5.1. Egész állandók

Egy egész állandó számjegyek sorozatából áll, amit oktális számként értelmezünk, ha a `0`-val (nulla számjeggyel) kezdődik és decimális számként minden más esetben. Az oktális állandókban nem fordulhatnak elő a `8` és `9` számjegyek. A számjegyek `0x` vagy `0X` (nulla számjegy) kezdetű sorozatát hexadecimális egész számként értelmezzük. A hexadecimális számok számjegyei közé tartoznak a 10...15 értékű számjegyeket jelző a vagy `A` ... `f` vagy `F` karakterek.

Az egész állandók az `u` vagy `U` betűből álló utótaggal láthatók el, ami azt jelzi, hogy a szám előjel nélküli. Az `l` vagy `L` utótag szintén használható és `long` típust jelöl.

Az egész állandók típusa a leírási formától, az értéktől és az utótagtól függ. (Lásd még az A4. pontban az adattípusok tárgyalásánál!) Ha a leírt szám utótag nélküli, decimális szám, akkor a típusa az értéke által meghatározott `int`, `long int` vagy `unsigned long int` típusok közül az első megfelelő típus. Ha a leírt szám utótag nélküli, oktális vagy hexadecimális szám, akkor típusa az `int`, `unsigned int`, `long int`, `unsigned long int` típusok közül az első megfelelő típus. Ha az utótag `u` vagy `U`, akkor a típus `unsigned int` vagy `unsigned long int`. Ha az utótag `l` vagy `L`, akkor a típus `long int` vagy `unsigned long int`.

> Az egész állandók típusának kimunkálása a könyv első kiadása óta lényegesen javult. Az első kiadásban még csak a nagy egész számokhoz használható `long` típus szerepelt. Az `U` utótag bevezetése új.

A2.5.2. Karakteres állandók

A karakteres állandó egy vagy több aposztrófok (`'`) közé zárt karakterből áll. Az egyetlen karakterből álló karakteres állandó értéke a karakternek a végrehajtáskor érvényes gépi karakterkészletből vett számértéke. A több karakterből álló karakteres állandók értéke a megvalósítástól függ.

A karakteres állandókban nem szerepelhet a `'` vagy az újsor-karakter; azért hogy ezeket, valamint bizonyos más karaktereket ábrázolni tudjuk, a következő escape-sorozatok használhatók:

új sor	NL (LF)	\n
vízszintes tabulátor	HT	\t
függőleges tabulátor	VT	\v
visszalépés (backspace)	BS	\b
kocsi-vissza	CR	\r
lapemelés (formfeed)	FF	\f
hangjelzés (bell)	BEL	\a
backslash	\	\\
kérdőjel	?	\?
aposztróf	'	\'
idézőjel	"	\"
oktális szám	ooo	\ooo
hexadecimális szám	hh	\xhh

A `\ooo` escape-sorozat egy backslash karakterből és az azt követő 1, 2 vagy 3 oktális számjegyből áll, amely a kívánt karakter értékét határozza meg. Erre a legjobb példa a `\0` (amit nem követ további számjegy), ami a `NUL` karaktert jelenti. A `\xhh` escape-sorozat a backslash karakterből, az azt követő x betűből és az utána írt hexadecimális számjegyek-

ből áll, amelyek a kívánt karakter értékét határozzák meg. A beírt számjegyek száma nincs korlátozva, de ha a kapott karakterérték nagyobb, mint a legnagyobb karakterérték, akkor a hatás definiálatlan. Ha a gépi megvalósítás a `char` típust előjelesen kezeli, akkor az oktális vagy hexadecimális escape-sorozatok értéke előjel-kiterjesztéssel keletkezik, csakúgy, mint a kényszerített típuskonverziójú `char` típus esetén. Ha a \ karaktert követő karakter nem a fentiek egyike, akkor az eredmény definiálatlan.

Néhány gépi megvalósítás kiterjesztett karakterkészletet használ, amelyben a karakteres állandók nem ábrázolhatók `char` típussal. Az ilyen kiterjesztett karakterkészletű karakteres állandó az `L` előtaggal írható be, pl. az `L'x'` formában, és ezt az állandót széles karakteres állandónak nevezzük. Az ilyen állandók `wchar_t` típusúak, ami egy egész adattípus és az `<stddef.h>` standard headerben van definiálva. Ezt a típust a közönséges karakteres állandókhoz, ill. oktális vagy hexadecimális escape-sorozatokhoz lehet használni, de a hatás definiálatlan, ha a megadott érték nagyobb a `wchar_t` típussal ábrázolható legnagyobb értéknél.

> Az escape-sorozatok némelyike új, különösen a hexadecimális karakterleírás. A kiterjesztett karakterkészlet szintén új. Az Egyesült Államokban és Nyugat-Európában általánosan használt karakterkészlet kódolása illeszkedik a char típushoz, főleg az ázsiai nyelvekhez való illeszkedés igényelte a wchar_t típus bevezetését.

A2.5.3. Lebegőpontos állandók

A lebegőpontos állandó egy egészrészből, tizedespontból, egy törtrészből, egy `e` vagy `E` betűből, egy opcionálisan előjelezhető kitevőből, valamint az `f, F, l` vagy `L` egyikének megfelelő opcionális utótagból áll. Az egészrész vagy a törtrészt számjegyek sorozata alkotja. Az egészrész vagy a törtrész (de nem mind a kettő) hiányozhat, csakúgy, mint a tizedespont vagy az `e` és a kitevő (de az egyiknek léteznie kell). A lebegőpontos állandó típusát az utótag határozza meg: az `f` vagy `F` `float` típust, az `l` vagy `L` `long double` típust jelöl, minden más esetben a típus `double`.

> Az utótag alkalmazása lebegőpontos állandók esetén új.

A2.5.4. Felsorolt állandók

`int` típusú állandók felsorolásaként deklarált azonosítók. (Bővebben lásd az A8.4. pontban!)

A2.6. Karaktersorozat-állandók

Egy karaktersorozat állandó (stringállandó) idézőjelekkel határolt karaktersorozatból áll, mint pl. a `"..."`. A karaktersorozat „karakteres tömb" típusú, `static` tárolási osztályú (l. az A4. pontot) és az adott karakterekkel inicializált adat. Az azonos karaktersorozat-állandók a gépi megvalósítástól függően különbözhetnek, és ha a program megkísérli a karaktersorozat-állandó tartalmát megváltoztatni, akkor az eredmény definiálatlan.

A szomszédos karaktersorozat-állandók egyetlen karaktersorozattá konkatenálódnak. Bármely konkatenáció után egy `\0` végjel íródik a karaktersorozathoz, így a program a karaktersorozatot végignézve azonosíthatja annak végét. A karaktersorozat-állandók nem tartalmazhatnak újsor vagy idézőjel-karaktereket, ezek ábrázolására – csakúgy, mint a karakteres állandók esetén – a megfelelő escape-sorozatok használhatók.

Ahogy azt már a karakteres állandóknál elmondtuk, a karaktersorozat-állandóknál is az L előtagot kell használni a kiterjesztett karakterkészlet esetén, pl. `L"..."` formában. A széles karaktersorozat-állandók „`wchar_` elemek tömbje" típusúak. A közönséges és széles karaktersorozat-állandók konkatenálásának eredménye definiálatlan.

> Az, hogy az azonos karaktersorozat-állandóknak nem szükségképpen kell megegyezniük, valamint hogy a tartalmuk nem módosítható, új az ANSI szabványban, csakúgy, mint a szomszédos karaktersorozatok konkatenálása. A széles karaktersorozat-állandók bevezetése szintén új.

A3. A szintaxis jelölése

A kézikönyvben a szintaktikai kategóriákat dőlt betűkkel, a literálisokat és karaktereket a programoknál használt betűtípussal jelöljük. Az alternatív kategóriák általában külön sorban, listaszerűen felsorolva szerepelnek, ill. néhány esetben a hosszú felsorolást egy sorba írtuk és előtte az „egyike a(z)" kifejezést használtuk. Az opcionális szimbólumokat az "*opc*" index jelzi, mint pl. a

$$\{\ \textit{kifejezés}_{opc}\ \}$$

esetén, ami egy kapcsos zárójelek között elhelyezett elhagyható kifejezést jelöl. A szintaxist az A13. pontban foglaltuk össze.

> Eltérően a könyv első kiadásában szereplő szintaktikai leírástól, itt explicit módon megadtuk a kifejezés-operátorok precedenciáját és asszociativitását.

A4. Az azonosítók értelmezése

Az azonosítók vagy nevek többféle dologra vonatkozhatnak: kijelölhetnek függvényt, struktúracímkét, uniont, felsorolást, struktúra vagy uniontagot, felsorolt állandót, `typedef` utasítással létrehozott típusnevet, ill. objektumot. Egy objektum, amit néha változónak nevezünk, a tárolóban helyezkedik el és az értelmezése két fő attribútumtól, a *tárolási osztálytól* és a *típustól* függ. A tárolási osztály az azonosítóhoz rendelt tárterület élettartamát, a típus pedig az azonosított objektumban tárolt érték jelentését határozza meg. Egy névhez egy érvényességi tartomány és egy csatolás is tartozik. Az érvényességi tartomány megadja, hogy a név a program melyik részében ismert, a csatolás pedig meghatározza, hogy ugyanaz a név egy másik érvényességi tartományban ugyanazt az objektumot vagy függvényt jelenti-e vagy sem. Az érvényességi tartomány és a csatolás leírása az A11. pontban található.

A4.1. A tárolási osztály

Két tárolási osztályt különböztetünk meg: automatikust és statikust. A tárolási osztályt több kulcsszó határozza meg az objektum deklarációjának szövegkörnyezetével együtt. Az automatikus tárolási osztályú objektumok egy blokkon belül helyiek, vagy más néven lokálisak (l. az A9.3. pontot), és a blokkból való kilépéskor elvesznek. Egy blokkon belül szereplő deklaráció, ha a tárolási osztályt külön nem specifikáltuk vagy az `auto` specifikációt használtuk, automatikus tárolási osztályú objektumot hoz létre. A `register` specifikációval deklarált objektum szintén automatikus, és (ha ez lehetséges) a számítógép gyors elérésű regisztereiben tárolódik.

A statikus objektumok egy blokkra érvényes lokális vagy több blokkra érvényes külső (external) típusúak lehetnek, de mindkét esetben a függvényből vagy blokkból való kilépés és visszatérés közti időszakban is megőrzik az értéküket. Blokkon belül, beleértve a függvényen belüli blokkot is, a statikus objektum a `static` kulcsszóval deklarálható. Az összes blokkon kívül, a függvénydefiníciókkal azonos szinten deklarált objektumok mindig statikus tárolási osztályúak. A statikus objektumok egy adott fordítási egységre vonatkozóan lokálissá tehetők a `static` kulcsszó alkalmazásával. Az ilyen objektumokhoz *belső csatolás* tartozik. A statikus objektumok a tárolási osztály explicit megadása nélkül a teljes programra nézve globálisak, vagy az `extern` kulcsszó használatával tehetők globálissá. Az ilyen objektumokhoz *külső csatolás* tartozik.

A4.2. Alapvető adattípusok

A C nyelvben számos alapvető adattípus létezik. A B. Függelékben leírt `<limits.h>` standard header definiálja az egyes adattípusok helyi gépi megvalósításban érvényes legnagyobb és legkisebb értékét. A B. Függelékben megadott számok a szóba jöhető legkisebb nagyságrendet jelentik.

A karakterként (`char`) deklarált objektumok elegendően nagyok ahhoz, hogy a végrehajtó rendszer karakterkészletének bármely tagját tárolni tudják. Ha egy, a karakterkészletből vett eredeti karaktert egy `char` típusú objektumban tárolunk, akkor annak értéke megegyezik a karakter egész értékű kódjával és garantáltan nem negatív. Egy `char` típusú változóban más objektumok is tárolhatók, de ilyenkor a rendelkezésre álló értékkészlet, valamint az érték előjeles vagy előjel nélküli ábrázolásmódja a gépi megvalósítástól függ.

Az `unsigned char` típusúnak deklarált karakterek ugyanakkora tárterületet igényelnek, mint a közönséges karakterek, de mindig *nem negatív* értékűek. Az explicit módon előjeles karaktereket `signed char` típusúnak kell deklarálni, és természetesen ezek is ugyanakkora helyet igényelnek, mint az egyszerű karakterek.

> A könyv első kiadásában az `unsigned char` típus nem szerepelt, de általánosan használták. A `signed char` típus új.

A `char` típus mellett még háromféle egész adattípus, a `short int`, `int` és `long int` alkalmazható. Az egyszerű `int` típusú objektum mérete megegyezik a befogadó számítógép társzervezéséből adódó természetes alapegységgel, és a speciális igények kielégítéséről más méretek gondoskodnak. A hosszabb egészek legalább akkora tárolóhelyet foglalnak el, mint a rövidebbek, de a gépi megvalósítás az egyszerű egészeket egyenlővé teheti a rövid vagy a hosszú egészekkel. Az `int` típus, ha csak másképpen nem specifikáltuk, mindig előjeles értéket jelent.

Az előjel nélküli egészek az `unsigned` kulcsszóval deklarálhatók és kielégítik a moduló 2^n aritmetika szabályait (ahol n a gépi ábrázoláshoz használt bitek száma), így az előjel nélküli egészekkel végzett aritmetikai műveletek során túlcsordulás soha nem fordulhat elő. A nem negatív értékek halmaza egy előjeles objektumban is tárolható, mint az értékek részhalmaza és ezek az értékek előjel nélküli objektumban is tárolhatók. Ilyenkor az átfedő értékek ábrázolása azonos.

Az egyszeres pontosságú lebegőpontos (`float`), a kétszeres pontosságú lebegőpontos (`double`) és az extra pontosságú lebegőpontos (`long double`) adattípusok egymás szinonimái lehetnek, de egy, a listában hátrébb álló típus legalább olyan pontosságú, mint az előrébb álló.

A `long double` típus új. Az első kiadásban értelmezett `long float` típus egyenértékű a `double` típussal. A `long float` típusmegadás megszűnt.

A *felsorolások* egyedi, egész értékű adattípusok, amelyek a névvel ellátott állandók felsorolásával kapcsolatosak (l. az A8.4. pontot). A felsorolások egész adatként viselkednek, de elég általános, hogy a fordítóprogram figyelmeztető jelzést ad, ha egy megadott felsorolás típusú objektumot nem annak egy állandójához vagy annak egy kifejezéséhez rendelünk.

Mivel az eddig felsorolt objektumok mindegyike számként értelmezhető, ezért ezeket *aritmetikai adattípusoknak* nevezzük. A char, az előjeles vagy előjel nélküli összes int, valamint a felsorolt típusokat összefoglaló néven *egész adattípusoknak* nevezzük. A float, double és long double típusokat *lebegőpontos adattípusoknak* nevezzük.

A `void` típus egy üres értékkészletet specifikál. A `void` típust függvények visszatérési értékének típusjelzésére használjuk, és azt jelenti, hogy nem jön létre visszatérési érték.

A4.3. Származtatott adattípusok

Az alapvető adattípusokon kívül a származtatott adattípusoknak elvileg végtelen sok változata létezik, amelyek az alapvető adattípusokból az alábbi módon hozhatók létre:

- *tömbök,* amelyek adott típusú objektumok sorozatából állnak;
- *függvények,* amelyek adott típusú objektummal térnek vissza;
- *mutatók,* amelyek adott típusú objektumot címeznek;
- *struktúrák,* amelyek különböző típusú objektumok sorozatából állnak;
- *unionok,* amelyek a különböző típusú objektumok bármelyikét tartalmazhatják.

Általában az objektumok előállítására használt módszereket rekurzívan is alkamazhatjuk.

A4.4. Típusminősítők

Egy objektum típusa járulékos minősítőkkel rendelkezhet. Egy objektum `const` deklarálása azt jelzi, hogy az objektum értéke nem fog megváltozni. Az objektum `volatile` deklarációja az optimalizálásnál lényeges speciális tulajdonságokat jelez. Egyetlen minősítő sem befolyásolja az objektum értékkészletét vagy aritmetikai tulajdonságait. A minősítőket részletesen az A8.2. pontban tárgyaljuk.

A5. Az objektumok és a balérték

Egy *objektum* a tároló névvel kijelölt része, a *balérték* pedig az objektumra hivatkozó kifejezés. A balérték (lvalue) kifejezésre jó példa a megfelelő típusú és tárolási osztályú azonosító. Az operátorok balértéket eredményeznek pl. ha `E` egy mutató típusú kifejezés, akkor a `*E` balérték kifejezés arra az objektumra hivatkozik, amire az `E` mutat. A „balérték" elnevezés az `E1 = E2` értékadó kifejezésből származik, amelyben az `E1` bal oldali operandusnak balértéknek kell lennie. Az egyes operátorok tulajdonságainak tárgyalásakor mindig megadjuk, hogy a balérték operandust igényel vagy a balérték operandust állít elő.

A6. Típuskonverziók

Néhány operátor az operandusaitól függően valamelyik operandusát az egyik adattípusról a másik adattípusra alakítja. Ebben a pontban ismertetjük az ilyen típuskonverziók várható eredményét. A közönséges operátorok típuskonverziós igényeit az A6.5. pontban összegezzük, és ezt az egyes operátorok tárgyalásánál további információkkal egészítjük ki.

A6.1. Az egész-előléptetés

Egy karakter, egy rövid egész számot, vagy egy egész értékű bitmezőt (függetlenül attól, hogy előjeles vagy előjel nélküli értékűek) vagy egy felsorolt típus objektumát minden olyan kifejezésben használhatjuk, amelyben egész mennyiséget használhatunk. Ha egy `int` típusú mennyiség az eredeti típus összes értékét (teljes értékkészletét) ábrázolja, akkor az értéke `int` típusúvá konvertálódik, máskülönben pedig `unsigned int` típusúvá. Ezt a típuskonverziós folyamatot egész-előléptetésnek (promóciónak) nevezzük.

A6.2. Egészek konverziója

Bármely egész úgy konvertálódik egy adott előjel nélküli típussá, hogy megkeressük azt a legkisebb nem negatív értéket, amely az előjel nélküli típussal ábrázolható legnagyobb értéknél eggyel nagyobb modulussal kongruens az egész számmal. Kettes komplemens kódú számábrázolás esetén ez megfelel a balról csonkításnak, ha az előjel nélküli típus a keskenyebb, és az előjel nélküli érték nullákkal való feltöltésének és előjel-kiterjesztéssel előjelezett értéknek, ha az előjel nélküli típus a szélesebb.

Amikor bármely egész számot előjeles típusúvá alakítunk, akkor az értéke változatlan marad, ha az az új típusban ábrázolható. Minden más esetben az eredmény a gépi megvalósítástól függ.

A6.3. Egész és lebegőpontos mennyiségek

Egy lebegőpontos típusú érték egésszé alakításakor a törtrész mindenképpen elvész, és ha az eredmény nem ábrázolható az egész típussal, akkor a művelet viselkedése definiálatlan. Különösen fontos megjegyezni, hogy egy negatív lebegőpontos érték előjel nélküli egész típussá alakításának eredménye nincs specifikálva.

Amikor egy egész értéket alakítunk lebegőpontossá és az érték az ábrázolható tartományban van, de egzaktul nem ábrázolható, akkor az eredmény a következő nagyobb vagy az előző kisebb ábrázolható érték lehet. Ha az átalakítás eredménye kívül esik az ábrázolható számok tartományán, akkor a művelet eredménye definiálatlan.

A6.4. Lebegőpontos típusok konverziója

Ha egy kisebb pontosságú lebegőpontos értéket egy vele egyező vagy nagyobb pontosságú lebegőpontos típussá alakítunk, akkor az érték változatlan marad. Ha egy nagyobb pontosságú lebegőpontos értéket alakítunk kisebb pontosságúvá, és az érték belül van az ábrázolható számtartományon, akkor az eredmény a következő nagyobb vagy előző kisebb ábrázolható érték lehet. Ha az eredmény az ábrázolható számtartományon kívülre esik, akkor a művelet eredménye definiálatlan.

A6.5. Aritmetikai típuskonverziók

Számos operátor eredményez típuskonverziót és az átalakítás után kapott típus előállításának módja hasonló. Az alapelv az, hogy az operandusokat olyan közös típusra hozzuk, ami megegyezik az eredmény típusával. Ezt a sémát a *szokásos aritmetikai típuskonverziónak* nevezzük.

> *Ha* az egyik operandus `long double` típusú, *akkor* a másik is `long double` típusúvá alakul.
> *Különben, ha* az egyik operandus `double` típusú, *akkor* a másik is `double` típusúvá alakul.
> *Különben, ha* az egyik operandus `float` típusú, *akkor* a másik is `float` típusúvá alakul.
> *Különben* mindkét operandusra az egész előléptetés fog végrehajtódni, és *ha* az egyik operandus `unsigned long int` típusú, *akkor* a másik is `unsigned long int` típusúvá alakul.
> *Különben, ha* az egyik operandus `long int`, a másik `unsigned int` típusú, *akkor* a működés attól függ, hogy a `long int` típusú mennyiség ábrázolható-e az unsigned `int` összes értékével, teljes értékkészletével; *ha igen,* akkor az `unsigned int` típusú operandus `long int` típusúvá alakul; *ha nem,* akkor mindkét operandus `unsigned long int` típusúvá alakul.
> *Különben, ha* az egyik operandus `long int` típusú, *akkor* a másik is `long int` típusúvá alakul.
> *Különben, ha* az egyik operandus `unsigned int` típusú, *akkor* a másik is `unsigned int` típusúvá alakul.
> *Különben* mindkét operandus `int` típusú.

Itt két változás van. Az első, hogy az aritmetika a `float` típusú mennyiségekkel egyszeres pontosságú műveleteket végez a kétszeres pontosságú helyett. A könyv első kiadása szerint minden lebegőpontos aritmetikai művelet kétszeres pontosságú. A második, hogy ha egy rövidebb előjel nélküli típus egy nagyobb előjeles típussal kombinálódik, akkor az előjel nélküli jelleg nem terjed ki az eredményre. A könyv első kiadása szerint mindig az előjel nélküli jelleg dominál. Az új konverziós szabályok kissé bonyolultak, de a meglepetést valamennyire csökkenti, hogy előfordulhat olyan eset, amikor előjel nélküli mennyiség előjelessel kombinálódik. Váratlan eredmény jöhet létre akkor is, amikor egy előjel nélküli kifejezést hasonlítunk össze egy ugyanolyan méretű előjeles kifejezéssel.

A6.6. Mutatók és egész mennyiségek

Egy mutatóhoz hozzáadható vagy abból kivonható egy egész típusú kifejezés, és ilyen esetben az egész kifejezés úgy konvertálódik, ahogyan azt az összeadás operátorának leírásában (az A7.7. pontban) specifikáljuk.

Két, azonos tömbben, azonos típusú objektumot címző mutató kivonható egymásból és az eredmény a kivonás operátorának leírásában (az A7.7. pontban) specifikált módon egész mennyiséggé konvertálódik.

Egy 0 értékű egész típusú állandó kifejezés, vagy kényszerített típuskonverzióval `void *` típusúvá alakított kifejezés kényszerített típuskonverzióval, értékadással vagy összehasonlítással bármilyen típusú objektum mutatójává konvertálható. Ez a művelet egy null-mutatót eredményez, ami egyenlő bármely, ugyanilyen típushoz tartozó null-mutatóval, de nem egyenlő bármely függvényt vagy objektumot címző mutatóval.

Más, mutatókra vonatkozó típuskonverziók is megengedettek, de ezek értelmezése

függhet a gépi megvalósítástól. Az ilyen átalakításokat egy explicit típuskonverziós operátorral vagy kényszerített típuskonverzióval kell specifikálni.

Egy mutató átalakítható saját tárolásához elegendően nagy egész típussá, de a kívánt méret a gépi megvalósítástól függ. Az átalakítást végző leképező függvény szintén a gépi megvalósítástól függ.

Az egyik adattípus mutatója egy másik adattípus mutatójává alakítható, de az eredményül kapott mutató címzési hibát jelezhet, ha az átalakított mutató nem megfelelő tárillesztésű objektumra hivatkozik. Csak az garantálható, hogy egy adott objektumhoz tartozó mutató változtatás nélkül egy azonos vagy enyhébb tárillesztési feltételeket igénylő adattípus mutatóvá konvertálható vagy abból visszakonvertálható. Megjegyezzük, hogy a „tárillesztés" a gépi megvalósítástól függ, és a legkevésbé szigorú tárillesztési feltétel a `char` típusú objektumokra vonatkozik. Mint az A6.8. pontban majd részletesen tárgyaljuk, egy mutató mindig változtatás nélkül átalakítható `void *` típusúvá, ill. abból visszaalakítható.

Minden mutató átalakítható egy másik, azonos típusú, legfeljebb a típusminősítő meglétében vagy hiányában különböző objektumra hivatkozó mutatóvá (l. az A4.4. és A8.2. pontot). Ha az objektum típusához minősítőt is rendelünk, akkor az új mutató egyenértékű lesz a régivel, kivéve, hogy az új minősítőtől eredő korlátozások vonatkoznak rá. Ha a minősítőt elhagyjuk, akkor az alapul szolgáló objektumra továbbra is az aktuális deklarációjában szereplő minősítő marad érvényben a műveletek során.

Végül, adott függvényhez tartozó mutató átalakítható egy másik függvénytípus mutatójává. A függvényt az átalakított mutatón keresztül hívva a hatás a gépi megvalósítástól függ, viszont ha az átalakított mutatót visszaalakítjuk az eredeti típusára, akkor az eredmény azonos lesz az eredeti mutatóval kapott eredménnyel.

A6.7. A void típus

Egy `void` típusú objektum (nemlétező) értékét nem lehet semmiféle módon felhasználni, és sem explicit, sem implicit konverzióval nem alakítható semmiféle nem `void` típussá. Mivel egy `void` kifejezés egy nemlétező értéket jelöl, ezért egy ilyen kifejezést csak ott lehet használni, ahol nincs szükség értékre, pl. kifejezésutasításként vagy egy vessző operátor bal oldali operandusaként (l. az A9.2. és az A7.18. pontot).

Egy kifejezés kényszerített típuskonverzióval alakítható `void` típusúvá. Például egy kényszerített `void` típuskonverzió törli egy kifejezésutasításban szereplő függvényhívás értékét.

> A `void` típus nem szerepelt a könyv első kiadásában, de azóta általánosan használt típussá vált.

A6.8. A void típushoz tartozó mutatók

Egy objektumhoz tartozó bármilyen mutató információveszteség nélkül átalakítható `void *` típusúvá. Ha az eredményt visszaalakítjuk az eredeti mutatótípusra, akkor az eredeti mutatót kapjuk vissza. Eltérően az A6.6. pontban a mutató—mutató konverziókról írtaktól, amely általában egy explicit kényszerített típuskonverziót igényel, bármely mutató értékül adható `void *` típusú mutatónak, ill. értéket kaphat `void *` típusú mutatótól, valamint az így kapott mutatók összehasonlításban is szerepelhetnek.

> A `void *` típusú mutató ilyen értelmezése új; a korábbiakban a `char *` típusú mutató játszotta a generikus mutató szerepét. Az ANSI szabvány különösen előnyben részesíti a `void *` és objektumhoz tartozó mutatók kombinációját értékadó és relációs utasításokban, viszont más kevert típusú mutatóhasználat esetén explicit kényszerített típuskonverziót igényel.

A7. Kifejezések

A kifejezésekben előforduló operátorok precedenciája megegyezik a következő tárgyalási sorrenddel, amelyben a legmagasabb precedenciájú operátorral kezdjük a tárgyalást. Így pl. azokat a kifejezéseket, amelyek a + operátor operandusai (A7.7. pont) lehetnek, az A7.1. ... A7.6. pontokban definiáljuk. Az egyes pontokban leírt operátorok azonos precedenciájúak, és leírásuknál megadjuk a bal vagy jobb oldali asszociativitásukat is. A szintaktika A13. pontbeli leírásában összesítve is megadjuk az operátorok precedenciáját és asszociativitását.

Az operátorok precedenciája és asszociativitása teljesen specifikált, de a kifejezések kiértékelési sorrendje, néhány kivételtől eltekintve definiálatlan, különösen, ha a részkifejezések mellékhatásokat eredményeznek. Ezért azt az esetet kivéve, amikor egy operátor garantálja, hogy operandusai az előírt sorrendben értékelődnek ki, a gépi megvalósítás szabadon dönthet az operandusok tetszőleges kiértékelési sorrendje mellett, vagy minden sorrend nélkül, a leghatékonyabban végezheti a kiértékelést. Természetesen az egyes operátorok az operandusaik által képviselt értékeket úgy kombinálják, hogy az kompatíbilis legyen annak a kifejezésnek a szintaktikai elemzésével, amelyben előfordul.

> Ez a szabály érvényteleníti az előzőekben megadott, a matematikailag kommutatív és asszociatív operátorokat tartalmazó kifejezések átrendezhetőségének szabadságára vonatkozó szabályt, de a számítási asszociativitásnak teljesülnie kell. A változás csak a lebegőpontos számokkal a pontossági korlátjuk közelében végzett számításokat befolyásolja, ill. olyan esetben lényeges, ahol túlcsordulás fordulhat elő.

A nyelvben nincs definiálva a túlcsordulás kezelése, az osztás ellenőrzése vagy a kifejezések kiértékelése során fellépő más kivételes esetek kezelése. A legtöbb létező C megvalósítás az előjeles egész kifejezések kiértékelésekor vagy értékadáskor fellépő túlcsordulást figyelmen kívül hagyja, de ez a működési mód nem garantálható. A nullával való osztás és az összes lebegőpontos extra eset kezelése gépi megvalósításonként változik, néha nem standard könyvtári függvényekkel oldható meg.

A7.1. Mutatógenerálás

Ha egy kifejezés vagy részkifejezés típusa valamilyen *T* típusra *„T tömbje"*, akkor a kifejezés értéke a tömb első objektumát címző mutatóra, és a kifejezés típusa *„mutató T-re"* típusra változik. Ez a konverzió nem történik meg, ha a kifejezés az unáris `&` operátor, vagy a `++`, `--`, ill. `sizeof` operátor operandusa, vagy egy értékadó operátor bal oldali operandusa, vagy egy . operátor operandusa. Hasonló módon egy kifejezés *„T-vel visszatérő függvény"* típusúról *„T-vel visszatérő függvény mutatója"* típusra konvertálódik, kivéve azt az esetet, amikor az `&` operátor operandusa.

A7.2. Elsődleges kifejezések

Elsődleges kifejezés az azonosító, az állandó, a karaktersorozat vagy a zárójelbe tett kifejezés. Az elsődleges kifejezés szintaktikai leírása:

elsődleges_kifejezés:
azonosító
állandó
karaktersorozat
(kifejezés)

Egy azonosító elsődleges kifejezés, ha a később ismertetett módon megfelelően deklarálták. Ilyenkor az azonosító típusát annak deklarációjában határozták meg. Egy azonosító balérték, ha egy objektumra hivatkozik (l. az A5. pontot) és ha típusa aritmetikai, struktúra, union vagy mutató típus.

Az állandó elsődleges kifejezés, és típusa függ az alakjától, mint azt az A2.5. pontban már tárgyaltuk.

A karaktersorozat állandó szinten elsődleges kifejezés. Típusa eredetileg „`char` *elemek tömbje*" (vagy széles karakterekből álló karaktersorozat esetén „`wchar_t` *elemek tömbje*"), de az A7.1. szabályt követve ez rendszerint „`char`-*hoz tartozó mutatóra*" vagy „`wchar_t`-*hez tartozó mutatóra*" módosul, és az eredmény a karaktersorozat első karakterére hivatkozó mutató lesz. A konverzió egyes inicializálásoknál nem jön létre, erre vonatkozóan l. az A8.7. pontot.

Egy zárójelbe tett kifejezés elsődleges kifejezés, amelynek típusa és értéke megegyezik annak egyszerű (zárójel nélküli) kifejezéséhez tartozó típussal és értékkel. A zárójel jelenléte nincs hatással a kifejezés esetleges balérték szerepére.

A7.3. Utótagos kifejezések

Az utótagos (postfix) kifejezésekben az operátorok csoportosítása balról jobbra történik. Az utótagos kifejezések szintaktikai leírása:

utótagos_kifejezés:
elsődleges_kifejezés
utótagos_kifejezés [*kifejezés*]
utótagos_kifejezés (*argumentum_kifejezés_lista*$_{opc}$)
utótagos_kifejezés . *azonosító*
utótagos_kifejezés –> *azonosító*
utótagos_kifejezés ++
utótagos_kifejezés – –

argumentum-kifejezés_lista:
értékadó_kifejezés
argumentum-kifejezés_lista, értékadó_kifejezés

A7.3.1. Tömbhivatkozások

Egy szögletes zárójelbe tett kifejezéssel követett utótagos kifejezés egy utótagos kifejezést alkot, ami egy indexelt tömbre való hivatkozást jelöl. A két kifejezés egyikének *„T-hez tartozó mutató"* típusúnak kell lennie, ahol *T* bármilyen típus lehet, és a másiknak egész típusúnak kell lennie. Az indexkifejezés *T* típusú. Az `E1[E2]` kifejezés definíció szerint azonos a `*((E1)+(E2))` kifejezéssel. A kérdéssel bővebben az A8.6.2. pontban foglalkozunk.

A7.3.2. Függvényhívások

Egy függvényhívás szintén utótagos kifejezés, amelyet a hívott függvény nevét követő, zárójelben elhelyezett, vesszővel elválasztott elemekből álló (esetleg üres) értékadó kifejezés lista alkot. Az értékadó kifejezések listája képezi a függvény argumentumlistáját. Ha

az utótagos kifejezés az aktuális érvényességi tartományban nem deklarált azonosítóból áll, akkor az azonosító implicit módon úgy deklarálódik, mint ha az

```
extern int azonosító();
```

deklaráció a függvényhívást tartalmazó legbelső blokkban helyezkedne el. Az utótagos kifejezés típusának (az esetleges implicit deklaráció és az A7.1. pontban leírt mutatógenerálás után) *„T értékkel visszatérő függvény mutatója"* típusúnak kell lennie (ahol *T* bármilyen típus lehet), és a függvényhívás értéke *T* típusú.

> A könyv első kiadásában a típus mindössze a „függvény" megjelölésre korlátozódott, és a függvény mutatón keresztüli hívásához egy explicit * operátort kellett alkalmazni. Az ANSI szabvány támogatja néhány fordítóprogram azon gyakorlatát, amely megengedi a függvény és a mutatóval specifikált függvény azonos szintaktika szerinti hívását. Ennek ellenére a régi szintaktika még használható.

Az *argumentum* megjelölést a függvényhívással átadott kifejezésekre alkalmazzuk, a *paraméter* megjelölést pedig a függvény definíciójában átvett, ill. a deklarációjában leírt bemeneti objektumokra (vagy azok azonosítójára) használjuk. Néha ugyanilyen értelemben használjuk az „aktuális argumentum (paraméter)" és a „formális argumentum (paraméter)" fogalmakat is.

A függvényhívás előkészítése során az egyes argumentumokról másolat készül és minden argumentumátadás szigorúan érték szerint történik. A függvény megváltoztathatja a paraméterként átadott objektumok értékét, amik az argumentumkifejezés másolatai, de ez a változtatás semmiféle módon nem befolyásolhatja az argumentum értékét (vagyis a hívó oldalon szereplő értéket). Természetesen lehetőség van rá, hogy a függvénynek mutatót adjunk át és ekkor a függvény megváltoztathatja a mutatóval címzett eredeti (hívó oldali) objektum értékét is.

A függvényt kétféle stílusban deklarálhatjuk. Az új stílusú deklarációban a paraméterek típusát explicit módon adjuk meg és az a függvény típusmegadásának része. Az ilyen deklarációt függvényprototípusnak is nevezzük. A régi stílusú deklarációban a paraméterek típusát nem specifikáljuk. A függvénydeklaráció kérdéseit az A8.6.3. és A10.1. pontokban tárgyaljuk.

Ha a híváshoz tartozó függvénydeklaráció régi stílusú, akkor alapfeltételezés szerint az egyes argumentumokra bekövetkezik az ún. argumentum-előléptetés, vagyis az egész típusú argumentumok az A6.1. pontban leírt egészelőléptetéssel konvertálódnak, a `float` típusú argumentumok pedig `double` típusúvá alakulnak. A függvényhívás hatása definiálatlan, ha az argumentumok száma nem egyezik meg a definícióban szereplő paraméterek számával, vagy ha egy argumentum típusa az előléptetés után nem egyezik meg a megfelelő paraméter típusával. A típusegyeztetés attól függ, hogy a függvény definíciója régi vagy új stílusú. Ha a definíció régi stílusú, akkor a híváskor megadott argumentum előléptetett típusát hasonlítja össze a gép a paraméter előléptetett típusával. Ha a definíció új stílusú, akkor az argumentum előléptetett típusának meg kell egyezni a paraméter előléptetés nélküli típusával.

Ha a híváshoz tartozó függvénydeklaráció új stílusú, akkor az argumentumok típusa úgy konvertálódik, mint az értékadásnál, vagyis a függvényprototípusban szereplő megfelelő paraméterek típusára alakul át. Az argumentumok számának meg kell egyezni az explicit módon leírt paraméterek számával, kivéve azt az esetet, ha a deklaráció paraméterlistája a további ki nem írt paraméterekre utaló `(, ...)` jelzéssel végződik. Ilyen esetben az argumentumok száma egyenlő vagy több kell legyen, mint a paraméterek száma és az explicit módon beírt paraméterekhez képest többlet argumentumok az előzőekben leírt argumentum-előléptetésnek lesznek kitéve. Ha függvény definíciója

régi stílusú, akkor a híváskor a prototípusban látható egyes paraméterek típusának meg kell egyezni a definícióban szereplő, a definíció paramétereire végrehajtott argumentum előléptetés utáni paramétertípusokkal.

> Ezek a szabályok különösen bonyolultak, mivel az új és a régi stílusban deklarált függvények keveredésére vonatkoznak. A kétféle deklaráció kevert használatát, amennyiben lehetséges, kerüljük!

Az argumentumok kiértékelési sorrendje nincs meghatározva, a különböző fordítóprogramok eltérően viselkednek. Mindezek ellenére az argumentumok és maga a függvénykijelölés a függvénybe való belépés előtt teljesen kiértékelődnek, beleértve a mellékhatásokat is.

A7.3.3. Struktúrahivatkozások

Utótagos kifejezést alkot egy utótagos kifejezést követő pontból és az azt követő azonosítóból álló szerkezet. Az első operandust alkotó kifejezésnek sturktúrának vagy unionnak, a pont után következő azonosítónak pedig egy struktúra- vagy uniontag nevének kell lennie. Az így kapott utótagos kifejezés értéke a struktúra vagy union megnevezett tagjának értéke és típusa a tag típusa. A kifejezés balérték, ha az első kifejezés balérték és ha a második kifejezés típusa nem egy tömb.

Egy utótagos kifejezést követő nyílból (amit a - és > jelekből rakunk össze) és egy azt követő azonosítóból álló szerkezet szintén utótagos kifejezést alkot. Az első operandust alkotó kifejezésnek sruktúrát vagy uniont címző mutatónak, az azonosítónak pedig a truktúra- vagy uniontag nevének kell lennie. A művelet eredménye a mutatókifejezéssel címzett struktúra vagy union megnevezett tagjának értéke és típusa a tag típusának felel meg. Az eredmény balérték, ha a típus nem tömbtípus.

A fentiek alapján az `E1->TAG` kifejezés azonos a `(*E1).TAG` kifejezéssel. A sturktúrákat és az unionokat az A8.3. pontban ismertetjük.

> A könyv első kiadásában már szerepelt az a szabály, hogy a tag neve mint kifejezés az utótagos kifejezésben szereplő struktúrához vagy unionhoz tartozik. Mindenesetre ez a megjegyzésben elismert szabály nem volt következetesen érvényre juttatva. A jelenlegi fordítóprogramok és az ANSI szabvány már érvényre juttatja.

A7.3.4. Utótagos inkrementálás

Egy utótagos kifejezést követő ++ vagy -- operátorból álló szerkezet szintén utótagos kifejezés. A kifejezés értéke az operandus értéke. Az érték elővétele (és felhasználása) után az operandus értéke ++ esetén eggyel növekszik (inkrementálás), -- esetén pedig eggyel csökken (dekrementálás). Az operandusnak balértéknek kell lenni. Az operandusra vonatkozó további megszorítások, ill. a működés részletei az additív operátoroknál (A7.7.) és az értékadásnál (A7.17.) találhatók.

A7.4. Egyoperandusú operátorok

Az egyoperandusú (unáris) operátorokkal létrehozott kifejezések csoportosítása jobbról balra történik. Az egyoperandusú kifejezések szintaktikai leírása:

egyoperandusú_kifejezés:
utótagos_kifejezés
++ *egyoperandusú_kifejezés*
-- *egyoperandusú_kifejezés*
egyoperandusú_operátor kényszerített_típuskonverziójú_kifejezés
`sizeof` *egyoperandusú_kifejezés*
`sizeof` *(típusnév)*

egyoperandusú_operátorok: egyike a következőknek:
`&` `*` `+` `-` `~` `!`

A7.4.1. Előtagos inkrementáló operátorok

Egy egyoperandusú kifejezést megelőző ++ vagy -- operátorból álló szerkezet szintén egyoperandusú kifejezés. A végrehajtás során az operandus ++ esetén eggyel növekszik (inkrementálás), -- esetén pedig eggyel csökken (dekrementálás). A kifejezés értéke az inkrementálás vagy dekrementálás után kapott érték. Az operandusnak balértéknek kell lennie. Az operandusra vonatkozó további megszorítások, ill. a működés részletei az additív operátoroknál (A7.7.) és az értékadásnál (A7.17.) találhatók.

A7.4.2. Címoperátor

Az egyoperandusú `&` operátor az operandusának a címét állítja elő. Az operandusnak nem bitmezőre vagy `register` típusúnak deklarált objektumra hivatkozó balértéknek, vagy függvény típusúnak kell lenni. Az eredmény egy mutató, amely a balértékkel egy objektumot vagy függvényt címez. Ha az operandus típusa *T*, akkor az eredmény típusa *„T típust címző mutató"*.

A7.4.3. Indirekciós operátor

Az egyoperandusú * indirekciós operátor eredményül azt az objektumot vagy függvényt adja, amelyre az opreandusa mutat. Az eredmény balérték, ha az operandus egy aritmetikai objektum, struktúra, union vagy mutató típus mutatója. Ha a kifejezés típusa *„T típust címző mutató"*, akkor az eredmény típusa *T*.

A7.4.4. Egyoperandusú plusz operátor

Az egyopreandusú + operátor operandusának aritmetikai típusúnak kell lenni, és az eredmény az operandus értéke. Egy egész típusú operandusra végrehajtódik az egészelőléptetés. Az eredmény típusa megegyezik az előléptetett operandus típusával.

> Az egyoperandusú + új az ANSI szabványban. Azt az egyoperandusú - operátor miatt, szimmetria okokból vezették be.

A7.4.5. Egyoperandusú mínusz operátor

Az egyoperandusú – operátor operandusának aritmetikai típusúnak kell lenni, és az eredmény az operandus negáltja. Egész típusú operandus esetén végrehajtódik az egész-előléptetés. Egy előjel nélküli mennyiség negáltját úgy számítjuk ki, hogy az előléptetett értékét kivonjuk az előléptetett típussal ábrázolható legnagyobb számból, és hozzáadunk egyet. A negatív nulla értéke nulla lesz. A művelet eredményének típusa megegyezik az előléptetett operandus típusával.

A7.4.6. Egyes komplemens operátor

A ~ operátor operandusának egész típusúnak kell lenni és az eredmény az operandus egyes komplemense. Az egész-előléptetés itt is végrehajtódik. Ha az operandus előjel nélküli, akkor az eredményt úgy számítjuk ki, hogy az operandus értékét kivonjuk az előléptetett típussal ábrázolható legnagyobb számból. Ha az operandus előjeles mennyiség, akkor az eredményt úgy számítjuk ki, hogy az előléptetett operandust a megfelelő előjel nélküli típusra konvertáljuk, végrehajtjuk a ~ műveletet, majd a kapott értéket visszakonvertáljuk előjeles típusra. Az eredmény típusa megegyezik az előléptetett operandus típusával.

A7.4.7. Logikai negálás operátora

A ! operátor operandusának aritmetikai vagy mutató típusúnak kell lenni, és az eredmény 1, ha az operandus értéke 0, ill. 0 minden más esetben. Az eredmény típusa `int`.

A7.4.8. Sizeof operátor

A `sizeof` operátor az operandusaként megadott típusú objektum tárolásához szükséges bájtok számát határozza meg. Az operandus egy kifejezés (amely nem értékelődik ki) vagy egy zárójelben elhelyezett típusnév lehet. A `sizeof` operátort `char` típusra alkalmazva, az eredmény 1 lesz, tömbre alkalmazva pedig a tömb által lefoglalt teljes terület bájtban mért hossza. Az operátort struktúrára vagy unionra alkalmazva az eredmény az objektumban lévő bájtok száma, beleértve a helykitöltő bájtokat is (amelyek az illesztési feltételek teljesítése miatt szükségesek). Tömb esetén a hosszt úgy kapjuk meg, hogy az elemek számát szorozzuk egy elem méretével. Az operátort nem alkalmazhatjuk függvény típusra, bitmezőre vagy nem teljes típusra. Az eredmény előjel nélküli egész típusú állandó, amelynek a típusa a gépi megvalósítástól függ. Ezt a típust az `<stddef.h>` standard header (l. a B. Függeléket) `size_t` típusként definiálja.

A7.5. Kényszerített típusmódosító

Egy egyoperandusú kifejezést megelőző zárójelbe tett típusnév a kifejezés értékének megadott típusra alakítását okozza. A kényszerített típusmódosítás szintaktikai leírása:

kényszerített_típusmódosítójú_kifejezés:
    egyoperandusú kifejezés
    (típusnév) kényszerített_típusmódosítójú_kifejezés

Ezt a konstrukciót kényszerített típusmódosításnak (a C nyelv terminológiájában *cast* szerkezetnek) nevezik. A típusnevek leírása az A8.8. pontban található, az átalakítás hatásait pedig az A6. pontban adtuk meg. A kényszerített típusmódosítójú kifejezés nem balérték.

A7.6. Multiplikatív operátorok

A * , / és % multiplikatív operátorok csoportosítása balról jobbra történik. A szintaktikai leírásuk:

multiplikatív_kifejezés:
kényszerített_típusmódosítójú_kifejezés
multiplikatív_kifejezés * *kényszerített_típusmódosítójú_kifejezés*
multiplikatív_kifejezés / *kényszerített_típusmódosítójú_kifejezés*
multiplikatív_kifejezés % *kényszerített_típusmódosítójú_kifejezés*

A * és / operátorok operandusainak aritmetikai típusúnak, a % operátor operandusainak egész típusúnak kell lenni. A művelet során az operandusok szokásos aritmetikai átalakításai végbemennek és ezekből meghatározható az eredmény várható típusa.

A kétoperandusú * operátor a szorzás műveletét jelzi.

A kétoperandusú / operátor a hányadost, a kétoperandusú % operátor a maradékot állítja elő, ha az első operandust osztjuk a másodikkal. Ha a második operandus nulla, akkor az eredmény definiálatlan. Ha a második operandus nulla, akkor mindig igaz az

```
(a/b)*b + a%b = a
```

összefüggés. Ha mindkét operandus nem negatív, akkor a maradék nem negatív és mindig kisebb, mint az osztó. Ha a fenti feltétel nem teljesül, akkor csak az garantálható, hogy a maradék abszolút értéke kisebb az osztó abszolút értékénél.

A7.7. Additív operátorok

A + és – additív operátorok csoportosítása balról jobbra történik. Ha az operandusok aritmetikai típusúak, akkor végbemennek a szokásos aritmetikai típuskonverziók. Mindkét operátor esetén vannak további típuslehetőségek is. Az additív operátorok szintaktikai leírása:

additív_kifejezés:
multiplikativ_kifejezés
additív_kifejezés + *multiplikatív_kifejezés*
additív_kifejezés – *multiplikatív_kifejezés*

A + operátor eredménye az operandusok összege. Egy tömb egy objektumát címző mutató bármilyen egész típusú értékkel összeadható, és ilyenkor a második operandus a mutatóval címzett objektum méretével való szorzással egy címeltolássá alakul. Az eredmény az eredeti mutatóval azonos típusú mutató, amely ugyanazon tömb másik, az eredeti objektumtól a címeltolással odébb lévő objektumát címzi. Így ha `P` egy tömb objektumának mutatója, akkor `P + 1` kifejezés szintén egy mutató, amely a tömb következő objektumát címzi. Ha az összegként kapott mutató a tömb határain kívülre címez, kivéve a felső határ utáni első helyet, akkor az eredmény definiálatlan.

A tömb utolsó eleme utáni első elemre való hivatkozás lehetősége új. Ez a lehetőség legalizálja a tömbök ciklusban való feldolgozásánál használt szokásos programszerkezetet.

A – operátor eredménye az operandusok különbsége. Egy mutatóból bármilyen egész típusú érték kivonható, és a műveletre az összeadásnál elmondott típuskonverziós szabályok, ill. feltételek érvényesek.

Ha két, azonos típusú objektumot címző mutatót kivonunk egymásból, akkor az eredmény előjeles egész érték, ami két megcímzett objektum közötti címkülönbséget jelenti. Ez az érték egymást követő objektumok esetén 1. Az eredmény típusa a gépi megvalósítástól függ, de az `<stddef.h>` standard headerben ez a típus `ptrdiff_t` típusként van definiálva. A kivonással kapott érték csak akkor definit, ha a mutatók azonos tömb elemeit címzik. Ha P egy tömb utolsó elemére mutat, akkor mindig igaz, hogy

`(P+1)-P=1.`

A7.8. Léptető operátorok

A << és >> léptető operátorok csoportosítása balról jobbra történik. Mindkét operátor operandusainak egész típusúaknak kell lenni és végbemegy az egész-előléptetés. Az eredmény típusa megegyezik a bal oldali, előléptetett operandus típusával. Az eredmény definiálatlan, ha a jobb oldali operandus negatív, vagy ha értéke a bal oldali kifejezés típusának megfelelő gépi ábrázolás bitszámával egyenlő, ill. annál nagyobb. A szintaktikai leírás:

léptető_kifejezés:
additív_kifejezés
léptető_kifejezés << *additív_kifejezés*
léptető_kifejezés >> *additív_kifejezés*

Az `E1<<E2` kifejezés értéke a bitmintaként értelmezett `E1` `E2` számú bittel balra léptetett értéke, ami, ha nem jött létre túlcsordulás, akkor a 2^{E2} értékkel való szorzásnak felel meg. Az `E1>>E2` kifejezés értéke az `E1` `E2` számú bittel jobbra léptetett értéke. A jobbra léptetés 2^{E2} értékkel való osztásnak felel meg, ha `E1` előjel nélküli vagy nem negatív mennyiség. Minden más esetben az eredmény a gépi megvalósítástól függ.

A7.9. Relációs operátorok

A relációs operátorok csoportosítása balról jobbra történik, de ennek nincs túl nagy jelentősége, mivel a kiértékelés során az `a<b<c` reláció mindig `(a<b)<c` alakra íródik át és először mindig az `a<b` értékelődik ki, aminek értéke 0 vagy 1 lesz. A szintaktikai leírás:

relációs_kifejezés:
léptető_kifejezés
relációs_kifejezés < *léptető_kifejezés*
relációs_kifejezés > *léptető_kifejezés*
relációs_kifejezés <= *léptető_kifejezés*
relációs_kifejezés >= *léptető_kifejezés*

A < (kisebb), > (nagyobb), <= (kisebb vagy egyenlő) és >= (nagyobb vagy egyenlő) operátorok 0 eredményt adnak, ha a kijelölt reláció hamis és 1 eredményt, ha igaz. Az

eredmény `int` típusú. Az aritmetikai típusú operandusokon végrehajtódnak a szokásos aritmetikai típuskonverziók. Csak (a minősítőket figyelmen kívül hagyva) azonos típusú objektumokhoz tartozó mutatók hasonlíthatók össze és az eredmény a címzett objektumok címtartományon belüli egymáshoz viszonyított (relatív) helyétől függ. A mutatók összehasonlítása csak azonos objektum részeire van értelmezve: ha két mutató ugyanazon egyszerű objektumot címzi, akkor összehasonlíthatók egyenlőségre; ha a mutatók ugyanazon struktúra tagjait címzik, akkor a struktúrában később deklarált objektumokhoz tartozó mutatók összehasonlíthatók a nagyobb feltétel szerint; ha a mutatók egy tömb elemeit címzik, akkor az összehasonlítás egyenértékű a megfelelő indexek összehasonlításával. Ha a `P` mutató egy tömb utolsó elemét címzi, akkor az összehasonlításban `P+1` nagyobb mint `P`, függetlenül attól, hogy `P+1` már a tömbön kívülre mutat. Minden, itt felsoroltaktól eltérő esetben a mutatók összehasonlítása nincs definiálva.

> Ezek a szabályok azzal, hogy megengedik egy struktúra vagy union különböző tagjaihoz tartozó mutatók összehasonlítását, kissé liberalizálják az előző kiadásban megfogalmazott korlátozásokat. Az új szabályok a tömb utolsó utáni eleméhez tartozó mutatóval való összehasonlítást szintén legalizálják.

A7.10. Egyenlőségoperátorok

Az egyenlőségoperátorok szintaktikai leírása:

egyenlőség_kifejezés:
relációs_kifejezés
egyenlőség_kifejezés `==` *relációs_kifejezés*
egyenlőség_kifejezés `!=` *relációs_kifejezés*

A `==` (egyenlő valamivel) és a `!=` (nem egyenlő valamivel) operátorok megegyeznek a megfelelő relációs operátorokkal, kivéve, hogy alacsonyabb a precedenciájuk. (Így `a<b == c<d` akkor 1, ha `a<b` és `c<d` egyformán igaz vagy egyformán hamis.)

Az egyenlőségoperátorok eleget tesznek mindazon szabályoknak, mint a relációs operátorok, de azokhoz képest további lehetőségeket is megengednek: mutatót összehasonlíthatunk egy állandó egész kifejezéssel vagy egy `void` típushoz tartozó mutatóval (l. A6.6. pontot).

A7.11. Bitenkénti ÉS operátor

A szintaktikai leírás:

ÉS_kifejezés:
egyenlőség_kifejezés
ÉS_kifejezés `&` *egyenlőség_kifejezés*

A szokásos aritmetikai típuskonverziók végbemennek, az eredmény az operandusok bitenkénti ÉS (AND) kapcsolata. Az operátor csak egész típusú operandusokra alkalmazható.

A7.12. Bitenkénti kizáró VAGY operátor

Szintaktikai leírás:

kizáró_VAGY_kifejezés:
ÉS_kifejezés
kizáró_VAGY_kifejezés ^ *ÉS_kifejezés*

A szokásos aritmetikai típuskonverziók végrehajtódnak, az eredmény az operandusok bitenkénti kizáró VAGY kapcsolata. Az operátor csak egész típusú operandusokra alkalmazható.

A7.13. Bitenkénti inkluzív VAGY operátor

Szintaktikai leírás:

inkluzív_VAGY_kifejezés:
kizáró_VAGY_kifejezés
inkluzív_VAGY_kifejezés | *kizáró_VAGY_kifejezés*

A szokásos aritmetikai típuskonverziók végbemennek, az eredmény az operandusok bitenkénti inkluzív VAGY kapcsolata. Az operátor csak egész típusú operandusokra alkalmazható.

A7.14. Logikai ÉS operátor

Szintaktikai leírás:

logikai_ÉS_kifejezés:
inkluzív_VAGY_kifejezés
logikai_ÉS_kifejezés && *inkluzív_VAGY_kifejezés*

Az && operátor csoportosítása balról jobbra történik. A művelet eredménye 1, ha mindkét operandus nullától különböző, és 0 különben. Az & operátortól eltérően az && operátor esetén garantált a balról jobbra történő végrehajtás: először az első operandus értékelődik ki, beleértve az összes mellékhatást is, és ha ez 0 értékű, akkor a teljes kifejezés értéke nulla. Különben a jobb oldali operandus is kiértékelődik, és ha az értéke nulla, akkor a teljes kifejezés értéke nulla. Minden más esetben a teljes kifejezés értéke 1.

Nem szükséges, hogy az operandusok azonos típusúak legyenek, de mindegyiknek aritmetikai vagy mutató típusúnak kell lennie. A művelet eredménye `int` típusú.

A7.15. Logikai VAGY operátor

A szintaktikai leírás:

logikai_VAGY_kifejezés
logikai_ÉS_kifejezés
logikai_VAGY_kifejezés || *logikai_ÉS_kifejezés*

A || logikai VAGY operátor csoportosítása balról jobbra történik. A művelet eredménye 1, ha az egyik operandus nullától különböző és minden más esetben 0. Az | operátortól eltérően a || operátor esetén garantált a balról jobbra való kiértékelés: először mindig az első operandus értékelődik ki (beleértve a mellékhatásokat is) és ha ez nem egyenlő nullával, akkor a kifejezés értéke 1. Máskülönben a jobb oldali operandus értékelődik ki, és ha ez nem egyenlő nullával, a kifejezés értéke 1, ha nulla, akkor pedig 0.

Nem szükséges, hogy az operátor operandusai azonos típusúak legyenek, de mindegyiknek aritmetikai vagy mutató típusúnak kell lennie. A művelet eredménye `int` típusú.

A7.16. Feltételes operátor

A szintaktikai leírás:

feltételes_kifejezés:
logikai_VAGY_kifejezés
logikai_VAGY_kifejezés ? *kifejezés* : *feltételes_kifejezés*

A művelet végrehajtása során kiértékelődik az első kifejezés (beleértve a mellékhatásokat is) és ha ez nem nulla, akkor az eredmény a második kifejezés értéke, különben pedig a harmadik kifejezés értéke. A második és harmadik kifejezések közül mindig csak az egyik értékelődik ki. Ha a második és harmadik kifejezés (operandus) aritmetikai típusú, akkor a szokásos aritmetikai típuskonverziók mennek végbe a közös típus elérése érdekében, és ez a közös típus lesz az eredmény típusa is. Ha mindkét operandus `void` típusú, vagy azonos típusú struktúra, ill. union, vagy azonos típusú objektumok mutatója, akkor az eredmény típusa a közös típus. Ha az egyik operandus mutató, a másik pedig 0 értékű állandó, akkor a 0 mutató típusra konvertálódik és ez lesz az eredmény típusa is. Ha az egyik operandus `void` típushoz tartozó mutató, a másik pedig nem, akkor a másik operandus is `void` típushoz tartozó mutatóvá alakul és ez lesz az eredmény típusa is.

A mutatók típusának összehasonlításakor a mutatókkal címzett objektumok típusát meghatározó típusminősítők (l. A8.2. pont) érdektelenek, de az eredmény típusa örökli a feltétel mindkét oldalának típusminősítőjét.

A7.17. Értékadó kifejezések

A C nyelvben számos értékadó operátor létezik, amelyek csoportosítása jobbról balra történik. Az értékadó kifejezés szintaktikai leírása:

értékadó_kifejezés:
feltételes_kifejezés
egyoperandusú_kifejezés értékadó_operátor értékadó_kifejezés

értékadó_operátor: egyike a következőknek:
`= *= /= %= += -= <<= >>= &= ^= |=`

Az összes operátor bal oldali operandusként balértéket igényel és ennek a balértéknek módosíthatónak kell lennie, azaz nem lehet tömb és nem lehet nemteljes vagy függvény típusú. Az szintén követelmény, hogy a típusa nem lehet `const` minősítésű, ill. ha struktúra vagy union, akkor nem lehet egyetlen tagja vagy rekurzívan beleágyazott altagja

sem `const` minősítésű. Egy értékadó kifejezés típusa a bal oldali operandus típusával egyezik meg és értéke az értékadás után a bal oldali operandusban tárolt érték lesz.

Az egyszerű, = jellel történő értékadás esetén a kifejezés értéke beíródik a balértékkel kijelölt objektumba. A következő állítások egyike igaz kell hogy legyen: mindkét operandus aritmetikai típusú, és ebben az esetben a jobb oldali operandus az értékadás során a bal oldali operandus típusára konvertálódik; mindkét operandus azonos típusú struktúra vagy union; az egyik operandus mutató, a másik `void` típushoz tartozó mutató; a bal oldali operandus egy mutató és a jobb oldali egy 0 értékű állandó kifejezés; mindkét operandus függvényhez vagy azonos típusú objektumokhoz (kivéve, hogy a jobb oldali operandus nem lehet `const` vagy `volatile` minősítésű) tartozó mutató.

Az `E1` *op*`=` `E2` alakú kifejezés egyenértékű az `E1 = E1` *op* `(E2)` kifejezéssel, kivéve, hogy `E1` csak egyszer értékelődik ki.

A7.18. Vesszőoperátor

A szintaktika leírása:

kifejezés:
értékadó_kifejezés
kifejezés , *értékadó_kifejezés*

Egy vesszővel elválasztott kifejezéspár balról jobbra értékelődik ki és a bal oldali kifejezés értéke elvész. Az eredmény típusa és értéke megegyezik a jobb oldali operandus típusával és értékével. A jobb oldali operandus kiértékelése során létrejövő mellékhatások teljesen lezajlanak. Olyan szövegkörnyezetben, ahol a vessző speciális jelentésű pl. függvények argumentumlistájában (A7.3.2.) vagy inicializáló kifejezések listájában (A8.7.), az igényelt szintaktikai egység egy értékadó kifejezés, így a vesszőoperátor csak zárójelbe tett csoporton belül jelenhet meg. Például az

```
f(a, (t=3, t+2), c)
```

függvénynek három argumentuma van, és ezek közül a második (vesszőoperátorral előállított) értéke 5.

A7.19. Állandó kifejezések

Szintaktikailag egy állandó kifejezés az operátorok egy részhalmazára korlátozódó kifejezés. A szintaktikai leírása:

állandó_kifejezés:
feltételes_kifejezés

A kifejezéseket kiértékelve állandó értéket kapunk, ami különböző értelemben használható (pl. a `case` utasítás után, tömbhatárként vagy bitmező hosszaként, felsorolt állandó értékeként, kezdeti értékként, az előfeldolgozó rendszer bizonyos kifejezéseiben).

Az állandó kifejezések nem tartalmazhatnak értékadást, inkrementáló vagy dekrementáló operátorokat, függvényhívást vagy vesszőoperátort, kivéve a `sizeof` operátor operandusát. Ha az állandó kifejezésnek egész típusúnak kell lennie, akkor az operandusai csak egész, felsorolt, karakteres és lebegőpontos állandókat tartalmazhatnak, a kénysze-

rített típuskonverziónak egész típust kell kijelölni és bármely lebegőpontos állandót kényszerített típuskonverzióval kell egész típusúvá alakítani. Ennek következtében a műveletben nem szerepelhetnek tömbökre vonatkozó műveletek, indirekció, címgenerálás és struktúratagra vonatkozó művelet. (Viszont bármely operandusra alkalmazható a `sizeof` operátor.)

A C nyelv a kezdeti értéket adó állandó kifejezések számára sokkal tágabb lehetőségeket enged meg. Az operandusok bármilyen típusú állandók lehetnek, az egyoperandusú `&` operátor alkalmazható a külső vagy statikus objektumokra, valamint állandó kifejezéssel indexelt statikus tömbökre. Az egyoperandusú `&` operátor ugyancsak alkalmazható implicit módon indexeletlen tömbökre és függvényekre. A kezdeti értéket adó kifejezés kiértékelésével állandót vagy egy korábban deklarált külső vagy statikus objektum állandóval növelt vagy csökkentett címét kell kapnunk.

Az `#if` utasítás utáni egész típusú állandó kifejezésekre kevesebb lehetőség van megengedve: ilyen helyen a `sizeof` kifejezés, felsorolt állandó és kényszerített típuskonverzió nem alkalmazható (l. az A12.5. pontot).

A8. Deklarációk

A deklarációk határozzák meg a fordítóprogram számára az egyes azonosítók értelmezését. A deklaráció nem szükségszerűen jelent tárbeli helyfoglalást az azonosító számára. A tárterületet lefoglaló deklarációkat definíciónak nevezzük. A deklaráció formája:

deklaráció:
deklaráció_specifikátorok kezdeti_deklarátor_lista$_{opc}$;

A kezdeti deklarátorlistában szereplő deklarátorok tartalmazzák a deklarálandó azonosítókat. A deklarációspecifikátorok típus és tárolási osztályspecifikátorokból állnak.

deklaráció_specifikátorok:
tárolási_osztály_specifikátor deklaráció_specifikátor$_{opc}$
típus_specifikátor deklaráció_specifikátor$_{opc}$
típus_minősítő deklaráció_specifikátor$_{opc}$

kezdeti_deklarátor_lista:
kezdeti_deklarátor
kezdeti_deklarátor_lista , kezdeti_deklarátor

kezdeti_deklarátor:
deklarátor
deklarátor = kezdeti_érték

A deklarátorokat az A8.5. pontban fogjuk részletesen tárgyalni, itt csak annyit említünk meg, hogy a deklarátorok tartalmazzák a deklarálandó neveket. Egy deklarációnak legalább egy deklarátort vagy egy struktúracímkét, unioncímkét, ill. felsorolástagot deklarálótípus specifikátort kell tartalmaznia. Az üres deklarációk nem használhatók.

A8.1. Tárolásiosztály-specifikátorok

A tárolásiosztály-specifikátorok a következők:

tárolási_osztály_specifikátor:
```
auto
register
static
extern
typedef
```

Az egyes tárolási osztályok jelentését az A4. pontban tárgyaltuk.

Az `auto` és `register`specifikátorok azt mondják ki, hogy a deklarált objektumok automatikus tárolási osztályúak és csak függvényeken belül használhatók. Az ilyen deklarációk egyben definíciók is és lefoglalják az objektum számára a tárolóhelyet. Egy `register` deklaráció egyenértékű az `auto` deklarációval, de arra utal, hogy a deklarált objektumot gyakran kívánjuk használni. Ténylegesen csak kevés objektum helyezkedik el regiszterben és csak meghatározott típusú objektumok lehetnek regiszteres típusúak. A korlátozások a gépi megvalósítástól függenek. Ha az objektum `register` tárolási osztályúnak deklarált, akkor az egyoperandusú `&` címgeneráló operátor sem explicit, sem implicit módon nem alkalmazható rá.

> Új az a szabály, hogy a `register` tárolási osztályúnak deklarált, de ténylegesen `auto` tárolási osztályú objektum címét nem szabad kiszámítani.

A `static` specifikáció hatására a deklarált objektum statikus tárolási osztályú lesz, és függvények belsejében vagy azokon kívül egyaránt használható. Egy függvény belsejében a `static`specifikátor egyben tárterületet is rendel az objektumhoz (azaz definícióként szolgál), a függvényen kívüli alkalmazásra vonatkozóan l. az A11.2. pontot.

Az `extern` deklaráció használható a függvények belsejében és azt jelzi, hogy a deklarált objektumhoz máshol rendeljük hozzá a tárterületet, a függvényen kívüli alkalmazásokra vonatkozóan l. az A11.2. pontot.

A `typedef`specifikátor nem foglal le tárterületet és csak a kényelmes szintaktikai leírás miatt nevezzük tárolásiosztály-specifikátornak. A részletes leírása az A8.9. pontban található.

Egy deklarációban legfeljebb csak egy tárolásiosztály-specifikátor adható meg. Ha egyet sem adunk meg, akkor a következő szabályok érvényesek: egy függvényen belül deklarált objektum mindig `auto` tárolási osztályú lesz; a függvényen belül deklarált függvények mindig `extern` tárolási osztályúak lesznek; a függvényen kívül deklarált objektumok és függvények mindig külső csatolással rendelkező statikus tárolási osztályúak lesznek. A részletesebb leírás az A10. és A11. pontokban található.

A8.2. Típusspecifikátorok

A típusspecifikátorok a következők:

típus_specifikátor:
```
void
char
short
int
```

```
long
float
double
signed
unsigned
```

struktúra_vagy_union_specifikátor
felsorolás_specifikátor
typedef_név

A `short` vagy `long` specifikátorok közül legfeljebb az egyik alkalmazható az `int` specifikátorral együtt, és az ilyen deklarációk jelentése ugyanaz, mint ha az `int` nem is szerepelne benne. A `long` specifikátor alkalmazható a `double` specifikátorral együtt is. A `signed` vagy `unsigned` specifikátorok közül legfeljebb egy alkalmazható az `int`-tel vagy annak `short`, ill. `long` változatával, vagy `char` specifikátorral együtt. A `signed` és `unsigned` specifikátorok bármelyike önmagában is megjelenhet, az `int` megadása magától értetődő, így elhagyható. A `signed` specifikátor alkalmazásával a `char` típusú mennyiségekre rákényszeríthető az előjeles adatábrázolás, viszont az alkalmazásuk az egész típusokkal együtt megengedett, de redundáns.

A fentieken kívül, minden más esetben egy deklarációban legfeljebb csak egy típusspecifikátor használható. Ha egy deklarációból hiányzik a típusspecifikátor, akkor a fordítóprogram a deklarált objektumot `int` típusúnak tekinti.

A típusok minősítettek is lehetnek, és a minősítés a deklarált objektum speciális tulajdonságait jelzi.

típusminősítő:
```
const
volatile
```

A típusminősítők bármelyik típusspecifikátorral együtt is megjelenhetnek. A `const` minősítésű objektumhoz kezdeti értéket rendelhetünk, de azután az értéke már nem változtatható. A `volatile` minősítőjű objektumoknak nincs a gépi megvalósítástól független szemantikájuk.

> A `const` és `volatile` minősítővel jelzett tulajdonságok újak az ANSI szabványban. A `const` minősítő az objektumról kinyilvánítja, hogy az egy csak olvasható memóriában (ROM-ban) is elhelyezhető és ezzel talán növeli az optimalizálhatóság lehetőségeit. A `volatile` minősítő szerepe, hogy a gépi megvalósítást az egyébként alkalmazott optimalizálások elhagyására kényszerítse. Például egy tárleképezéses bemenetet–kimenetet használó számítógép esetén egy perifériaregiszterhez tartozó mutató `volatile` minősítésű mutatóként deklarálható, azért, hogy a fordítóprogram ne távolítsa el a nyilvánvalóan redundáns (de szükséges) mutatón keresztüli hivatkozásokat. Eltekintve a `const` minősítőjű objektumokhoz való explicit változtatási kísérletek diagnosztizálásától, a fordítóprogram ezeket a minősítőket figyelmen kívül hagyhatja.

A8.3. Struktúrák és unionok deklarációja

A struktúra különböző típusú, névvel azonosított tagok sorozatából álló objektum. Az union különböző időpontokban a különböző típusú tagok egyikét tartalmazza. A struktúra- és unionspecifikátorok azonos alakúak.

struktúra_vagy_union_specifikátor:
struktúra_vagy_union azonosító$_{opc}$ { *struktúra_deklarációs_lista* }
struktúra_vagy_union azonosító

struktúra_vagy_union:
`struct`
`union`

A struktúradeklarációs lista a struktúra vagy union tagjaihoz tartozó deklarációk sorozata.

struktúra_deklarációs_lista:
struktúra_deklaráció
struktúra_deklarációs_lista struktúra_deklaráció

struktúra_deklaráció:
specifikátor_minősítő_lista struktúra_deklarátor_lista;

specifikátor_minősítő_lista:
típus_specifikátor specifikátor_minősítő_lista$_{opc}$
típusminősítő specifikátor_minősítő_lista$_{opc}$

struktúra_deklarátor_lista:
struktúra_deklarátor
struktúra_deklarátor_lista , struktúra_deklarátor

Általában egy struktúra deklarátor a sturktúra vagy union egy tagjának deklarátora. Egy struktúra tag meghatározott számú bitből is állhat és az ilyen tagokat bitmezőnek vagy röviden mezőnek nevezzük. A mező hossza a deklarátorból, a mező nevét követő kettőspont utáni részből vehető ki.

struktúra_deklarátor:
deklarátor
deklarátoropc : *állandó_kifejezés*

A következő alakú típusspecifikátor

struktúra_vagy_union azonosító { *struktúra_deklarációs_lista* }

az azonosítót a listában megadott struktúra vagy union címkéjeként deklarálja. Egy soron következő, az előzővel azonos vagy azon belüli érvényességi tartományú deklaráció ugyanerre a típusra a specifikátorban használt címke alapján, a lista nélkül hivatkozhat. Ennek formája:

struktúra_vagy_union azonosító

Ha egy specifikátor a címkével, de lista nélkül jelenik meg, amikor a címke még nincs deklarálva, akkor egy nemteljes típus megadásáról beszélünk. Egy nemteljes struktúra vagy union típusú objektum minden olyan programkörnyezetben használható, ahol az objektum méretére nincs szükség, és csak ilyen esetekben használható. Megengedett pl. a használata deklarációkban (de nem definíciókban), mutató specifikálásakor vagy `typedef` létrehozásakor. Ügyeljünk arra, hogy a listával ellátott specifikátorban a struk-

túra vagy union típusa a listán belül nemteljes deklarációjú és csak akkor válik teljessé, ha a specifikátort a } kapcsos zárójellel lezárjuk!

Egy struktúra nem tartalmazhat nemteljes típusú tagot, ezért lehetetlen olyan struktúrát vagy uniont deklarálni, amely saját magát tartalmazza. Ettől függetlenül a struktúra vagy union típushoz nevet rendelhetünk és az így kapott címke már lehetővé teszi önhivatkozó struktúrák definiálását. Ez azon alapszik, hogy a struktúra vagy union tartlamazhat egy önmagát címző mutatót, mivel a nemteljes típusokhoz is deklarálhatók mutatók.

Nagyon speciális szabály alkalmazható a

struktúra_vagy_union azonosító ;

alakú deklarációkra, amelyek egy struktúrát vagy uniont deklarálnak, de nincs a deklarációban sem deklarációs lista, sem deklarátor. Ez a deklaráció az aktuális érvényességi tartományon belül még akkor is létrehoz egy új, nemteljes típusú struktúrát vagy uniont az azonosítónak megfelelő címkével, ha az azonosító egy külső érvényességi tartományban már deklarált struktúra vagy union címkéje.

> Ez a mélyértelmű szabály új az ANSI szabványban. Feladata, hogy gondoskodjon a belső érvényességi tartományban deklarált olyan kölcsönösen rekurzív struktúrák kezeléséről, amelyek címkéje már deklarálva volt a külső érvényességi tartományban.

Egy listát igen, de címkét nem tartalmazó struktúra- vagy unionspecifikátor egy egyedi típust hoz létre, amire közvetlenül csak abban a deklarációban hivatkozhatunk, amelynek része.

A tagok és címkék neve nem kerül konfliktusba egymással vagy a közönséges változók nevével. Egy tagnév nem jelenhet meg kétszer ugyanazon struktúrában vagy unionban, de ugyanaz a tagnév más struktúrában vagy unionban használható.

> A könyv első kiadásában a struktúra- vagy uniontagok neve nem volt kapcsolatban a „szülővel", viszont ez a kapcsolat már az ANSI szabvány megjelenése előtt általánossá vált a fordítóprogramokban.

Egy struktúra vagy union nem mező típusú tagja bármilyen objektumnak megfelelő típussal rendelkezhet. Egy mező típusú tag (amelyhez nem szükséges, hogy deklarátor tartozzon, ezért név nélküli is lehet) `int`, `unsigned int` vagy `signed int` típusú, és úgy értelmezhető, mint az adott számú bitnek megfelelő hosszúságú egész típusú objektum. Az, hogy az `int` típusú mező, mint előjeles mennyiség, hogyan kezelődik, a gépi megvalósítástól függ. A struktúra szomszédos, mező típusú tagjai a gépi megvalósítástól függő méretű tárolóhelyekre, egymás mellé kerülnek, de az elhelyezési sorrendjük szintén a megvalósítástól függ. Amikor egy mezőt követő másik mező nem illeszthető egy részlegesen feltöltött tárolóhelybe, akkor a fordítóprogram megosztja azokat két tárolóhely között vagy a második mezőt teljes egészében egy új tárolóhelyre teszi és a részlegesen feltöltött tárolóhelyre helykitöltő egységet rak. Ezt a helykitöltést a 0 hosszúságú, név nélküli mező alkalmazásával lehet kikényszeríteni, így az utána következő mező már biztosan a következő tárolóhely kezdetére kerül.

> Az ANSI szabvány a mezőket sokkal inkább függővé tette a gépi megvalósítástól, mint ahogy az a könyv előző kiadásában volt; ezért minden kritika nélkül célszerű a bitmezők tárolási szabályait mint a nyelv „gépi megvalósítástól függő" részét olvasni. A bitmezőket tartalmazó struktúrák a struktúra tárolóhelyigényének csökkentésére alkalmas hordozható megoldásként (bár ez valószínűleg a bitmezők bonyolultabb kezelése miatt a futási idő növekedését eredményezi) vagy a bit-szinten ismert tárolóterület nem hordozható leírásaként használhatók. A második esetben az adott gépi megvalósítás szabályainak ismeretére van szükség.

Egy struktúra tagjai a deklaráció sorrendjében, folyamatosan növekvő címeken helyezkednek el. A nem mező típusú tagok az adott típustól függő címhatárhoz illeszkednek, ezért a struktúrában név nélküli lyukak (üres helyek) lehetnek. Ha egy struktúrát címző mutatót kényszerített típuskonverzióval a struktúra első tagját címző mutatóvá alakítunk, az eredő mutató ténylegesen a struktúra első tagjára fog mutatni.

Az uniont olyan struktúraként értelmezhetjük, amelynek tagjai a 0 ofszetnél kezdődnek és méretük elegendő bármelyik tag befogadására. Egy union egy időben legfeljebb csak egyetlen tagját tartalmazhatja. Ha egy uniont címző mutatót kényszerített típuskonverzióval a tagját címző mutatóvá alakítunk, akkor az eredményül kapott mutató magára a tagra mutat.

A következőkben egyszerű példát mutatunk a struktúra deklarálására.

```
struct tcsomo {
    char tszo[20];
    int szam;
    struct tcsomo *bal;
    struct tcsomo *jobb;
};
```

Ez a deklaráció egy 20 elemű karakteres tömbből, egy egészből és két, hasonló struktúrát címző mutatóból áll. Ha egyszer ezt a deklarációt megadtuk, akkor a

```
struct tcsomo s, *sp;
```

deklaráció a megadott fajtájú `s` struktúrát, valamint a megadott fajtájú struktúrát címző `sp` mutatót deklarálja. Ezeket a deklarációkat felhasználva az

```
sp->szam
```

kifejezés az `sp` mutatóval címzett struktúra `szam` nevű tagjára hivatkozik. Hasonló módon az

```
s.bal
```

az `s` struktúra bal oldali részfájának mutatójára hivatkozik. Az

```
s.jobb->tszo[0]
```

az `s` struktúrában lévő jobb oldali részfa `tszo` karakteres tömbjének első karakterét címzi.

Általában egy union tagját nem lehet ellenőrizni, kivéve, ha az unionhoz ugyanezen tagnak megfelelő értéket rendelünk. Ezért az unionok használatát egy speciális szabály egyszerűsíti: ha egy union többféle, de azonos kezdeti résszel (az elején azonosan deklarált tagokkal) rendelkező struktúrát tartalmaz és ha az union aktuális tartalma ezen struktúrák egyike, akkor megengedett, hogy a bármely, az unionban lévő ilyen struktúra közös kezdeti részére hivatkozzunk. Például a következő egy szintaktikailag helyes programrészlet:

```
union {
    struct {
        int tipus;
```

```
    } n;
    struct {
        int tipus;
        int intcsomo;
    } ni;
    struct {
        int tipus;
        float floatcsomo;
    } nf;
} u;
...
u.nf.tipus = FLOAT;
u.nf.floatcsomo = 3.14;
...
if (u.n.tipus == FLOAT)
... sin(u.nf.floatcsomo) ...
```

A8.4. Felsorolások

A felsorolás olyan egyedi típus, amelynek értékei sorra felveszik a névvel megadott állandók (elemek) halmazából a megfelelő értéket. A felsorolásspecifikátor alakját a struktúrák és unionok specifikátorától kölcsönözték. A szintaktikai leírás:

felsorolás_specifikátor:
enum *azonosító*$_{opc}$ { *felsorolás-lista* }
enum *azonosító*

felsorolás-lista:
felsorolt_érték
felsorolás_lista , *felsorolt_érték*

felsorolt_érték:
azonosító
azonosító = *állandó_kifejezés*

A felsorolás-listában lévő azonosítók `int` típusú állandóként vannak deklarálva, és a programban bárhol megjelenhetnek, ahol állandó kifejezésre van szükség. Ha nincs egyenlőségjel és az azt követő felsorolt érték, akkor a megfelelő állandók értéke nullával kezdődő, eggyel növekvő számsorozat lesz, amely hozzárendelődik a deklarációban szereplő azonosítókhoz. A hozzárendelés balról jobbra történik. Az = jelből és az utána következő felsorolt értékből álló lista esetén az azonosítóhoz a megadott érték rendelődik, és ha a listában soron következő azonosítóhoz nem tartozik = felsorolt érték rész, akkor az utolsó értékadástól folytatólagosan rendelődnek az értékek az azonosítókhoz.

A felsorolásban deklarált neveknek az adott érvényességi tartományon belül különbözniük kell egymástól és a közönséges változók neveitől, de az értékek megegyezhetnek.

A felsorolásspecifikátorban az azonosítók feladata analóg a struktúraspecifikátorban szereplő struktúracímkével vagyis megnevezi az adott felsorolást. A címkével és listával ellátott vagy anélküli felsorolásspecifikátorra vonatkozó szabályok ugyanazok, mint a struktúra- vagy unionspecifikátorokra vonatkozóak, kivéve, hogy nemteljes felsorolás

típus nem létezik. A felsoroláslista nélküli felsorolásspecifikátor címkéjének egy, az érvényességi tartományon belüli, listával ellátott specifikátorra kell hivatkozni.

A felsorolás a könyv első kiadása óta bevezetett új adatfajta, de már évek óta a C nyelv részét alkotja.

A8.5. Deklarátorok

A deklarátorok szintaktikája:

deklarátor:
mutató$_{opc}$ *direkt_deklarátor*

direkt_deklarátor:
azonosító
(deklarátor)
direkt_deklarátor [*állandó_kifejezés*$_{opc}$]
direkt_deklarátor (*paraméter_típus_lista*)
direkt_deklarátor (*azonosító_lista*$_{opc}$)

mutató:
típus_minősítő_lista$_{opc}$
típus_minősítő_lista$_{opc}$ *mutató*

típus_minősítő_lista:
* *típus_minősítő*
* *típus_minősítő_lista típus_minősítő*

A deklarátorok szerkezete hasonlít az indirekció, függvény és tömbkifejezések szerkezetéhez; a csoportosítás ugyanaz.

A8.6. A deklarátorok jelentése

A deklarátorok listája a típus- és tárolásiosztály-specifikátorok sorozata után jelenik meg. Minden egyes deklarátor egy egyedi fő azonosítót deklarál, ez az első alternatíva a direkt deklarátor szintaktikai leírásában. Erre az azonosítóra közvetlenül alkalmazzuk a tárolási- osztály-specifikátorokat, de a típus a deklarátor alakjától függ. Egy deklarátor azt jelenti, hogy amikor az azonosítója megjelenik a deklarátorral azonos alakú kifejezésben, akkor az a megadott típusú objektumot eredményezi.

Egyelőre foglalkozzunk csak a deklarációspecifikátor (A8.2.) típusleíró részével: egy adott deklarátor esetén a deklaráció `TD` alakú, ahol `T` a típus és `D` a deklarátor. A típus a különböző alakú deklarátorokban hozzárendelődik az azonosítóhoz, így ezt a jelölésrendszert használjuk a deklarátorok leírására.

Egy `TD` alakú deklarációban, ahol `D` sima azonosító, az azonosító típusa `T` lesz.

Egy `TD` alakú deklarációban, ahol `D`

```
( D1 )
```

alakú, a `D1`-ben szereplő azonosító típusa ugyanaz, mint `D`-ben. A zárójelzés nem változtatja meg a típust, csak az összetett deklarátorok kötésére lehet hatással.

A8.6.1. Mutatódeklarátorok

A `T D` deklarációban, ahol `D`

`*` *típusminősítő_lista*$_{opc}$ `D1`

alakú és az azonosító típusa a `T D1` deklarációban *„típusmódosított* `T`", a `D` azonosítójának típusa *„típus_módosító típus_minősítő_lista* mutató `T` típushoz" lesz. A minősítőt követő `*` operátor a mutatóra magára és nem a mutatóval címzett objektumra vonatkozik.

Például nézzük a következő deklarációt:

```
int *ap[];
```

Itt `ap[]` játssza a `D1` szerepét. Az `int ap[]` deklaráció az `ap`-hoz *„egész elemek tömbje"* típust rendel, a típusminősítő lista üres, és a típusmódosító *„tömbje a ...-nek"*. Így az aktuális deklaráció `ap`-hez „`int` *típusú adatokat címző mutatók tömbje*" típust rendel.

Egy másik példa a deklarációra:

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

Ez az első részében egy `i` egészt és a `pi` egészhez tartozó mutatót deklarál. A `cpi` állandó mutató értéke nem változhat, mindig ugyanarra a helyre fog mutatni és értéke nem módosítható (bár kezdeti érték hozzárendelhető, csakúgy, mint itt). A `pci` típusa „`const int-et` *címző mutató*", és a `pci`-t magát meg lehet változtatni, hogy más címre mutasson, de az értéket, amelyre mutat a `pci` mutatón keresztüli értékadással nem lehet módosítani.

A8.6.2. Tömbdeklarátorok

A `T D` deklarációban, ahol `D`

`D1 [` *állandó_kifejezés*$_{opc}$ `]`

alakú és az azonosító típusa a `T D1` deklarációban *„típusmódosított* `T`", a `D` azonosítójának típusa *„típusmódosított tömbje a* `T`-*nek*" lesz. Ha az állandó kifejezés jelen van, annak egész típusúnak és nullánál nagyobb értékűnek kell lennie. Ha az állandó kifejezéssel specifikált tömbhatár hiányzik, akkor a tömb nemteljes típusú.

Egy tömb előállítható aritmetikai típusból, mutatóból, struktúrából vagy unionból, vagy más tömbből (ez többdimenziós tömböt eredményez). Bármilyen típusú elemekből is állítottuk elő a tömböt, annak teljes típusúnak kell lenni, nem szabad, hogy nemteljes típusú tömbből vagy struktúrából álljon. Ezért egy többdimenziós tömbnek csak az első dimenziója hiányozhat. A nemteljes típusú tömb objektumának típusa egy másik, az objektumra vonatkozó teljes deklarációval (A10.2.) vagy kezdetiérték-adással (A8.7.) tehető teljessé. Például a

```
float fa[17], *afp[17];
```

deklaráció `float` típusú számokból álló tömböt és `float` típusú számokat címző mutatókból álló tömböt deklarál. Hasonlóan a

```
static int x3d[3][5][7];
```

deklaráció egy statikus, háromdimenziós egészekből álló tömböt deklarál, amelynek mérete 3*5*7 elem. Részleteiben nézve `x3d` valójában háromelemű tömb, amelynek minden eleme öt tömb tömbje, és ez utóbbi tömbök hét egész elem tömbjét alkotják. Az `x3d, x3d[i], x3d[i][j], x3d[i][j][k]` kifejezések bármelyike megjelenhet más kifejezésben, és ezek közül az első három kifejezés tömb típusú, a negyedik pedig `int` típusú. Pontosabban nézve az `x3d[i][j]` hét egész számból álló tömb, az `x3d[i]` pedig öt, egyenként hét egész számból álló tömb tömbje.

Az `E1[E2]` formában definiált tömbindexelési művelet azonos a `*(E+E2)` művelettel, ezért az aszimmetrikus megjelenés ellenére az indexelés kommutatív művelet. Mivel a konverziós szabályokat alkalmazni kell a + műveletre és a tömbökre (A6.6., A7.1., A7.7.), ha `E1` tömb és `E2` egész típusú, akkor `E1[E2]` az `E1` tömb `E2`-dik elemére hivatkozik.

A példánkban `x3d[i][j][k]` egyenértékű a `*(x3d[i][j] + k)`-vel. Az `x3d[i][j]` első részkifejezés az A7.1. pontban leírtak szerint „egészekből álló tömb mutatója" típusúvá alakítódik és az A7.7. szerint az összeadás egy egész méretének megfelelő többszörözést von maga után. Az eddigiek a tömb soronkénti tárolásának szabályából következnek. A deklarációban az első index segít a tömb által igényelt teljes tárolóterület meghatározásában, de nincs további szerepe az index kiszámításában.

A8.6.3. Függvénydeklarátorok

Az új stílusú függvénydeklaráció is felírható `T D` alakban, ahol `D`

`D1 (`*paraméter_típus_lista*`)`

alakú, és a `T D1` deklarációban levő azonosító „*típusmódosított* `T`" típusú, a `D`-ben szereplő azonosító pedig „*típusmódosított* függvény *paraméter_típus_lista* argumentumokkal és `T` típusú visszatérési értékkel" típusú.

A paraméterek szintaxisa:

paraméter_típus_lista:
paraméterlista
paraméterlista `, ...`

paraméter_lista:
paraméter_deklaráció
paraméterlista `,` *paraméter_deklaráció*

paraméter_deklaráció:
deklaráció_specifikátorok deklarátorok
deklaráció_specifikátorok absztrakt_deklarátor$_{opc}$

Az új stílusú deklarációban a paraméterlista meghatározza a paraméterek típusát. Speciális esetben az új stílusú függvény deklarátora nem tartalmaz paramétereket és a paraméterlistában mindössze a `void` kulcsszó áll. Ha a paraméterlista a „ `, ...`" résszel végződik, akkor a függvény a paraméterlistában explicit módon megadottakon kívül további argumentumokhoz is hozzáférhet (l. az A7.3.2. pontot).

A függvény tömb vagy függvény típusú paramétereinek típusa mutató típusra változik

az A10.1. pontban leírásra kerülő paraméterkonverziós szabályoknak megfelelően. Egy paraméter deklarációjában csak a `register` tárolásiosztály-specifikátor használható, és ez a specifikátor is törlődik, kivéve ha a függvénydeklarátor megelőzi a függvénydefiníciót. Hasonló módon, ha a paraméterdeklarációban a deklarátorok azonosítót tartalmaznak, valamint a függvénydeklarátor nem előzi meg a függvénydefiníciót, akkor az azonosítók kilépnek a pillanatnyi érvényességi tartományból. Az azonosítót nem tartalmazó absztrakt deklarátorokat az A8.8. pontban tárgyaljuk.

A régi stílusú függvénydeklaráció `T D` alakú felírásában `D`

`D1 (`*azonosító_lista*$_{opc}$`)`

formájú, és a `T D1` deklarációban szereplő azonosító típusa „*típusmódosított* `T`", amíg a `D`-ben szereplő azonosító típusa „*típusmódosított* függvény nem specifikált argumentumokkal, `T` típusú visszatérési értékkel" típus lesz. A paraméterek (ha jelen vannak) alakja:

azonosító_lista:
azonosító
azonosító_lista, **,** *azonosító*

A régi stílusú deklarátorokban az azonosítólistának hiányozni kell, kivéve, ha a deklarátort a függvénydefiníció előtt használjuk (l. A10.1.). Így a deklaráció alapján a paraméterek típusáról nincs információnk.

Példák függvénydeklarációra:

```
int f(), *fpi(), (*pfi)();
```

Ez egész értékkel visszatérő `f` függvényt, egészt címző mutatóval visszatérő `fpi` függvényt, valamint egész értékkel visszatérő függvényhez tartozó `pfi` mutatót deklarál. Egyik deklarációnál sem specifikáltuk a paraméter típusát, így ezek régi stílusú deklarációk.

A következőkben új stílusú deklarációt mutatunk be:

```
int strcpy(char *cel, const char *forras), rand(void);
```

Itt az `strcpy` egész értékkel visszatérő függvény, amelynek két argumentuma van és az első karaktert címző mutató, a második pedig állandó karaktereket címző mutató. A paraméterek neve egyben utal a szerepükre is. A másodiknak deklarált `rand` függvénynek nincs argumentuma és `int` értékkel tér vissza.

A C nyelv ANSI szabvány bevezetéséből adódó messze legfontosabb változása a paraméterprototípust tartalmazó függvénydeklarátorok alkalmazása. Ezeknek számos előnye van a könyv első kiadásában ismertetett „régi stílusú" deklarátorokkal szemben, mivel lehetővé teszik a hibaellenőrzést és a függvényhíváson keresztül korlátozzák az argumentumokat. Természetesen a bevezetésüknek ára is volt: a bevezetésekor fellépő zűrzavar és keveredés, valamint a kétféle formához való alkalmazkodás szükségessége. A kompatibilitás kedvéért néhány nem túl szép szintaktikai megoldásra volt szükség, mint pl. a `void` explicit jelzésként történő bevezetésére az új stílusú, paraméter nélküli függvények esetén.
A paraméterlista végén elhelyezhető „ `, ...` " kiegészítéssel jelzett változó hosszúságú paraméterlistájú függvények bevezetése szintén új. Ezt az `<stdarg.h>` standard headerben lévő makrókkal együtt megvalósított formális kezelési mechanizmust a könyv első kiadása hivatalosan tiltotta, de nem hivatalosan szemet hunyt felette.
Ezeket az új megoldásokat a szabvány a C++ nyelvből vette át.

A8.7. Kezdetiérték-adás

Egy objektum deklarálása során a kezdetiérték-deklarátorral adhatunk kezdeti értéket a deklarált azonosítónak. A kezdeti értéket = jelnek kell megelőzni, és az egy kifejezés vagy kapcsos zárójelek között elhelyezett kezdetiérték-lista lehet. A lista a megfelelő formátum kialakítása érdekében végződhet vesszővel. A szintaktikai leírás:

kezdeti_érték:
értékadó_kifejezés
{ *kezdeti_érték-lista* }
{ *kezdetiérték-lista* , }

kezdetiérték-lista:
kezdeti érték
kezdetiérték-lista , *kezdeti érték*

A statikus objektumokhoz vagy tömbökhöz tartozó kezdeti értékekben szereplő kifejezéseknek állandó kifejezéseknek kell lennie (ahogy ezt az A7.19. pontban leírtuk). Az `auto` vagy `register` tárolási osztályú objektumok vagy tömbök kezdetiérték-kifejezésének ugyancsak állandó kifejezésnek kell lennie, ha a kezdetiértékek kapcsos zárójelben lévő listában helyezkednek el. Viszont ha egy automatikus tárolási osztályú objektum kezdeti értéke egyetlen kifejezés, akkor nem szükséges, hogy az állandó kifejezés legyen (de az objektumhoz való hozzárendelés miatt a típusának megfelelőnek kell lennie).

> A könyv első kiadásában leírt C nyelv nem támogatta az automatikus tárolási osztályú struktúrák, unionok vagy tömbök kezdetiérték-adását. Az ANSI szabvány megengedi ezt, de csak állandó értékű konstrukcióval (kivéve, ha a kezdetiérték egyszerű kifejezéssel adható meg).

Egy nem explicit módon inicializált statikus objektum úgy inicializálódik, mintha önmaga vagy tagja állandó nulla értéket kapott volna. A nem explicit módon inicializált automatikus tárolási osztályú objektumok kezdeti értéke definiálatlan.

Egy mutató vagy egy aritmetikai típusú objektum kezdeti értéke egyetlen, esetleg kapcsos zárójelek között elhelyezett kifejezés. A kifejezés értéke hozzárendelődik az objektumhoz.

Egy struktúra kezdeti értéke vagy egy azonos típusú kifejezés, vagy egy kapcsos zárójelek között elhelyezett, a tagok sorrendjében felsorolt kezdeti értékekből álló lista lehet. A név nélküli bitmező típusú struktúra-tagokat a fordítóprogram figyelmen kívül hagyja és azok nem kapnak kezdeti értéket. Ha a listában kevesebb kezdeti érték van, mint a tagok száma, akkor a további tagok `0` kezdeti értéket kapnak. A tagok számánál több kezdeti érték nem lehet a listában.

Egy tömb inicializálása kapcsos zárójelben elhelyezett, az egyes elemekhez rendelt kezdeti értékek listájával történhet. Ha a tömb mérete ismeretlen, akkor a kezdeti értékek leszámolásával határozza meg a fordítóprogram a tömb méretét, és a típus így válik teljessé. Ha a tömb rögzített méretű, akkor a listában megadott kezdeti értékek száma nem haladhatja meg a tömb elemeinek számát. Ha a kezdeti értékek száma kisebb, mint a tömbelemek száma, akkor a további tömbelemek `0` kezdeti értéket kapnak.

Speciális esetként egy karakteres tömb karaktersorozattal inicializálható. Ilyenkor a karaktersorozat egymást követő karaktereit rendeli a fordítóprogram a tömb soron következő eleméhez. Hasonló módon egy széles karaktersorozat-állandóval (A2.6.) inicializálható a `wchar_t` típusú karakteres tömb. Ha a tömb méretét nem ismerjük, akkor a karaktersorozatban lévő és hozzá rendelődő karakterek száma (beleértve a karaktersorozatot lezáró null-karaktert is) határozza meg a tömb méretét. Ha a tömb mérete rögzített, akkor a karaktersorozatban lévő karakterek száma (nem számítva a lezáró null-karaktert) nem haladhatja meg a tömb deklarált méretét.

Egy union egy azonos típusú egyedi kifejezéssel vagy egy kapcsos zárójelbe tett kezdeti értékkel inicializálható, de az inicializálás mindig csak az union első tagjára alkalmazható.

> A könyv első kiadása még nem engedte meg az unionok inicializálását. Az „első tag" inicializálhatóságára bevezetett új szabály elég ügyetlen, de új szintaxis nélkül nehezen általánosítható. Azonkívül, hogy az ANSI szabvány megengedi az unionok explicit inicializálását, még megadja a nem explicit módon inicializált statikus unionok definit szemantikáját is.

Egy *aggregátum* olyan összetett objektum, amely struktúra vagy tömb jellegű. Ha egy aggregátum további aggregátum típusú tagokat tartalmaz, akkor az inicializálási szabályok rekurzívan alkalmazhatók. Az inicializálásból a kapcsos zárójelek elhagyhatók a következő szabályok alapján: ha egy aggregátum tagjához, amely maga is aggregátum, tartozó kezdeti értékek bal oldali kapcsos zárójellel kezdődnek, akkor a következő, kezdeti értékek vesszővel elválasztott sorozatából álló lista a részaggregátumok tagjait inicializálja. Ha a kezdeti értékek száma nagyobb a tagok számánál, akkor a fordítóprogram hibát jelez. Amennyiben a rész-aggregátumokhoz tartozó kezdetiérték-lista nem bal oldali kapcsos zárójellel kezdődik, akkor a fordítóprogram a listából csak a részaggregátum tagjai számának megfelelő elemet vesz figyelembe és a listában fennmaradó elemek azon aggregátum további tagjait fogják inicializálni, amelynek a részaggregátum a része volt.

Az elmondottakra nézzünk néhány példát! Az

```
int x[] = { 1, 3, 5 };
```

deklarálja az `x` egydimenziós tömböt és egyben inicializálja is azt. Mivel a tömb mérete nincs megadva és három elem kap kezdeti értéket, így a tömb háromelemű lesz. A

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

egy teljesen kapcsos zárójelezésű inicializálás: az 1, 3 és 5 érték inicializálja az `y[0]` tömb első sorát, azaz az `y[0][0]`, `y[0][1]` és `y[0][2]` elemeket. A következő két értéksor hasonló módon inicializálja az `y[1]` és `y[2]` tömböket. Mivel a kezdeti értékek a szükségesnél hamarabb fogynak el (a kezdeti értékek száma kisebb, mint a tömbelemek száma), ezért az `y[3]` tömb elemei 0 kezdeti értéket kapnak. Pontosan ugyanez a hatás érhető el a

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

inicializálással. Itt az `y` kezdeti értékeit tartalmazó lista kapcsos zárójellel kezdődik, de az `y[0]` tömbhöz tartozó lista nem. Ezért a listából három elem kerül felhasználásra. A továbbiakban a következő három elem az `y[1]` tömbhöz, az utolsó három elem pedig az `y[2]` tömbhöz rendelődik hozzá. `y` további elemei 0 kezdeti értéket kapnak. A

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

deklaráció y első oszlopát (y-t kétdimenziós tömbként értelmezve) inicializálja és a fennmaradó elemekhez 0 kezdeti értéket rendel.

Végül nézzük a következő példát: a

```
char msg[] = „Szintaktikai hiba a következő sorban: %s\n";
```

az msg karakteres tömb elemeit inicializálja egy karaktersorozattal. A tömb méretét a karaktersorozat hossza határozza meg, beleszámítva a karaktersorozatot lezáró nullkaraktert is.

A8.8. Típusnevek

Néhány összefüggésben (kényszerített típuskonverzióval specifikált explicit típusmódosításban, függvénydeklarátorokban a paraméterek típusának deklarálásakor, a sizeof argumentumaként alkalmazásakor) szükség lehet az adattípus nevének megadására. Ez a típusnév felhasználásával érhető el. A típusnév szintaktikailag egy adott típusú objektum olyan deklarációja, amelyből hiányzik az objektum neve. A szintaktikai leírás:

típusnév:
specifikátor_minősítő_lista absztrakt_deklarátor$_{opc}$

absztrakt_deklarátor:
mutató
mutató$_{opc}$ *direkt_absztrakt_deklarátor*

direkt_absztrakt_deklarátor:
(*absztrakt_deklarátor*)
direkt_absztrakt_deklarátor$_{opc}$ [*állandó_kifejezés*$_{opc}$]
direkt_absztrakt_deklarátor$_{opc}$ (*paraméter_típus_lista*$_{opc}$)

Az absztrakt deklarátorban egyértelműen azonosítható az a hely, ahol az azonosító megjelenhetne, ha a szerkezet egy deklaráción belüli deklarátor lenne. Az így megnevezett típus ilyenkor ugyanaz lesz, mint a hipotetikus azonosító típusa. Néhány példa:

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

Ezek a szerkezetek sorban egymás után „egész", „egészhez tartozó mutató", „három, egészekhez tartozó mutatóból álló tömb", „nem meghatározott számú egész elemből álló tömbhöz tartozó mutató", „nem meghatározott paraméterlistájú, egészhez tartozó mutatóval visszatérő függvény" és „paraméterlista nélküli, egész értékkel visszatérő függvényekhez tartozó mutatókból álló, nem meghatározott méretű tömb" típusokat neveznek meg.

A8.9. A typedef

Azok a deklarációk, amelyekben a tárolásiosztály-specifikátor a `typedef`, nem objektumot deklarálnak, hanem egy olyan azonosítót definiálnak, ami a későbbiekben típusnévként használható. Az így definiált azonosítókat *typedef neveknek* nevezzük. A szintaktikai leírás:

typedef_név
    azonosító

A `typedef` deklaráció az egyes, deklarátorokban szereplő nevekhez a szokásos módon (l. A8.6. pontot) hozzárendel egy típust. Ezért az ilyen `typedef` nevek szintaktikailag egyenértékűek a típusjelző kulcsszóval a megfelelő típushoz rendelt típusnévvel. Például a

```
typedef long Blokkszam, *Blokkptr;
typedef struct { double r, theta; } Complex;
```

deklarációk után a

```
Blokkszam b;
extern Blokkptr bp;
Complex z, *zp;
```

konstrukciók teljesen legális deklarációk lesznek. A `b` típusa `long`, így a `bp` egy „long típushoz tartozó mutató"; a `z` egy meghatározott struktúra, `zp` pedig ezt a struktúrát kijelölő mutató.

A `typedef` nem vezet be új típust, csak a más módon megadott típusok szinonimáit állítja elő. Például az előzőekben deklarált `b` ugyanolyan típusú, mint bármilyen más `long` típusú objektum.

A `typedef` nevek deklarálhatók a belső érvényességi tartományban, de nem üres típusspecifikátor-halmazt kell megadnunk. Például az

```
extern Blokkszam;
```

nem deklarálja a `Blokkszam`-ot, de az

```
extern int Blokkszam;
```

már igen.

A8.10. Típusekvivalenciák

Két típusspecifikátor-lista egyenértékű, ha mindegyik a típusspecifikátorok azonos halmazát tartalmazza, figyelembe véve, hogy ugyanazt a specifikátort más módon is megadhatjuk (pl. a `long` ugyanazt jelenti, mint a `long int`). A különböző címkéjű struktúrák, unionok és felsorolások különbözőek, és egy címke nélküli struktúra, union vagy felsorolás egy egyedi típust specifikál.

Két típus azonos, ha az absztrakt deklarátoruk (A8.8.) az esetleges `typedef` típusok kifejtése és bármilyen függvényparaméter azonosító törlése után ekvivalens típusspecifikátor-listákat eredményez. A tömbméretek és a függvényparaméter-típusok a típusekvivalencia meghatározásánál lényegesek.

A9. Utasítások

Az utasítások a leírásuk sorrendjében hajtódnak végre, kivéve azt, ahol külön jelezzük. Az utasítások végrehajtása a hatásukban nyilvánul meg és nem rendelkeznek értékkel. Az utasítások számos csoportba sorolhatók, és általános szintaktikai leírásuk:

utasítás:
címkézett_utasítás
kifejezés_utasítás
összetett_utasítás
kiválasztó_utasítás
iterációs_utasítás
vezérlésátadó_utasítás

A9.1. Címkézett utasítások

Az utasításokhoz előtagként megadott címke tartozhat. A címkézett utasítások szintaktikája:

címkézett_utasítás
azonosító : *utasítás*
`case` *állandó_kifejezés* : *utasítás*
`default` : *utasítás*

A címke egy azonosítóként deklarált azonosítóból áll. Egy azonosító címkét csak a `goto` utasítás célpontjaként használhatunk. Az azonosító címke érvényességi tartománya az akutális függvény (az a függvény, amelyben előfordul). Mivel a címkékhez nem tartozik megnevezett tárterület, ezért nem kerülhetnek kapcsolatba más azonosítókkal és nem deklarálhatók újra (l. az A11.1. pontot is).

A `case` és `default` címkéi a `switch` utasítással használhatók. A `case` utáni állandó kifejezésnek egész típusúnak kell lennie.

A címkék önmagukban nem módosítják az utasítások végrehajtásának sorrendjét.

A9.2. Kifejezésutasítások

Az utasítások többsége kifejezésutasítás, amelynek általános alakja:

kifejezésutasítás:
kifejezés$_{opc}$;

Funkcióját tekintve a legtöbb kifejezésutasítás értékadás vagy függvényhívás. A kifejezésutasításban lévő kifejezés összes mellékhatása lezajlik a következő utasítás végrehajtásának kezdete előtt. Ha kifejezés utasításból hiányzik a kifejezés, akkor ezt a konstrukciót null-utasításnak (üres utasításnak) nevezzük, és gyakran használjuk az iterációs utasítások üres ciklusmagjának helyettesítésére vagy címke helyének kijelölésére.

A9.3. Összetett utasítás

Vannak olyan programkörnyezetek, ahol a fordítóprogram csak egyetlen utasítást fogad el. Az összetett utasítás (vagy másnéven blokk) ennek a korlátozásnak a megszüntetését és több utasítás egyetlen utasításként kezelését teszi lehetővé. Például egy függvénydefiníció magja egyetlen összetett utasítás. Az összetett utasítás szintaktikai leírása:

összetett_utasítás:
{ *deklarációs_lista*$_{opc}$ *utasítás_lista*$_{opc}$ }
deklarációs_lista:
deklaráció
deklarációs_lista deklaráció

utasítás_lista:
utasítás
utasítás_lista utasítás

Ha a deklarációs listában található valamelyik azonosító a blokkon kívüli érvényességi körrel rendelkezik (a blokkon kívül már deklarálva van), akkor a külső deklaráció a blokkba való belépéskor fel lesz függesztve (l. az A11.1. pontot) és csak annak befejeztével nyeri vissza a hatályát. Egy blokkban egy azonosítót csak egyszer lehet deklarálni. Ezeket a szabályokat kell alkalmazni az összes, azonos névtérben lévő azonosítóra (A11.1.); a különböző névtérben lévő azonosítók egymástól különbözőként kezelhetők.

Az automatikus tárolási osztályú objektumok inicializálása a blokkba való minden egyes belépéskor, a blokk tetején megtörténik, és ugyanakkor sorban feldolgozza a program a deklarátorokat is. Ha kívülről egy vezérlésátadó utasítással a blokk belsejébe ugrunk, akkor ezek az inicializálások elmaradnak. A `static` tárolási osztályú objektumok csak egyszer, a program végrehajtásának kezdetén inicializálódnak.

A9.4. Kiválasztó utasítások

A kiválasztó utasítások minden esetben a lehetséges végrehajtási sorrendek egyikét választják ki. Általános szintaktikai leírásuk:

kiválasztó_utasítás:
`if (` *kifejezés* `)` *utasítás*
`if (` *kifejezés* `)` *utasítás* `else` *utasítás*
`switch (` *kifejezés* `)` *utasítás*

Az if utasítás mindkét formájában a kifejezés (amelynek aritmetikai vagy mutató típusú kifejezésnek kell lennie) kiértékelődik (beleértve az összes mellékhatást is) és ha az eredmény nem egyenlő nullával, akkor az első alutasítás hajtódik végre. Az `if` utasítás második alakja esetén a második alutasítás akkor hajtódik végre, ha a kifejezés nulla. Sokszor nem egyértelmű, hogy az `else` ág melyik `if` utasításhoz tartozik. Ezt a kétértelműséget a C nyelv azzal oldja fel, hogy egy `else` mindig az azonos blokkon belüli utolsó `else` nélküli `if` utasításhoz kötődik.

A `switch` utasítás hatására a vezérlés a kifejezés értékétől (amelynek egész típusúnak kell lennie) függően több utasítás egyikére adódik át. A `switch` utasítással vezérelt alutasítások tipikusan összetett utasítások. Az alutasításon belül bármely utasítást címkézhetünk egy vagy több `case` címkével (A9.1.). A végrehajtás során a vezérlő kifejezésre végbemegy az egészelőléptetés (l. A6.1.), és a `case` részek állandói az előléptetett típusra konvertálódnak. Ugyanazon `switch` utasításon belül két `case` rész állandójának a konverzió után nem lehet azonos értéke. Egy `switch` utasításhoz legfeljebb egy `default` címke is tartozhat. A `switch` utasítások egymásba ágyazhatók; a `case` és `default` címkék mindig ahhoz a legbelső `switch` utasításhoz kapcsolódnak, amely tartalmazza azokat.

A `switch` utasítás végrehajtásakor a kifejezés az összes mellékhatást beleértve kiértékelődik és összehasonlításra kerül az egyes `case` részek állandóival. Ha az egyik `case`

rész állandója megegyezik a kifejezés értékével, akkor a vezérlés átadódik a `case` címkét követő utasításra. Ha egyetlen `case` rész állandója sem egyezik a kifejezés értékével, és ha a `switch` utasítás tartalmaz `default` címkét, akkor a vezérlés a `default` címke utáni utasításra adódik át. Ha a `switch` utasításban nincs `default` rész és egyik `case` rész állandója sem egyezik meg a kifejezés értékével, akkor egyetlen alutasítás sem hajtódik végre.

> A könyv első kiadásában az a feltétel szerepelt, hogy a `switch` utasítás vezérlő kifejezésének és a `case` állandójának `int` típusúnak kell lenni, ez a szabványban úgy módosult, hogy csak egész jelleget követelnek meg.

A9.5. Iterációs utasítások

Az iterációs utasítások egy ciklust határoznak meg. Általános szintaktikai leírásuk:

iterációs_utasítás:
`while` (*kifejezés*) *utasítás*
`do` *utasítás* `while` (*kifejezés*) ;
`for` ($kifejezés_{opc}$; $kifejezés_{opc}$; $kifejezés_{opc}$) *utasítás*

A `while` és a `do` utasításban a program az alutasításokat ismételten végrehajtja mindaddig, amíg a kifejezés értéke nullától különböző marad. A kifejezésnek aritmetikai vagy mutató típusúnak kell lennie. A `while` utasítás esetén az ellenőrzés, beleértve a kifejezés kiértékelésekor adódó mellékhatásokat is, az utasítás végrehajtása előtt megy végbe, amíg a `do` utasítás esetén az ellenőrzés csak az egyes iterációk után történik meg.

A `for` utasításban az első kifejezés csak egyszer értékelődik ki és ez adja a ciklus kezdeti értékét. Az első kifejezés típusára vonatkozóan semmiféle megkötés nincs. A második kifejezésnek aritmetikai vagy mutató típusúnak kell lenni. Ez a kifejezés az egyes iterációk előtt értékelődik ki, és ha az értéke nullává válik, akkor a `for` ciklus befejeződik. A harmadik kifejezés szintén minden iteráció elején kiértékelődik és az így kapott érték adja a ciklus újbóli kezdő értékét (a ciklusváltozó aktuális értékét). Ennek típusára sincs semmiféle megkötés. Az egyes kifejezések kiértékelésekor adódó mellékhatások a kiértékelés után azonnal teljesen lecsengenek. Ha a ciklus alutasításai között nem szerepel a `continue`, akkor a

`for` (*1. kifejezés* ; *2. kifejezés* ; *3. kifejezés*) *utasítás*

szerkezetű ciklus egyenértékű az

```
1. kifejezés ;
while ( 2. kifejezés ) {
    utasítás
    3. kifejezés ;
}
```

szerkezetű ciklussal.

A három kifejezés bármelyike elhagyható. A második kifejezés hiánya esetén a `for` ellenőrző része úgy működik, mintha az ellenőrzés egy nem nulla értékű állandóval történne.

A9.6. Vezérlésátadó utasítások

A vezérlésátadó utasítások a vezérlés feltétel nélküli átadására alkalmasak. A szintaktikai leírásuk:

vezérlésátadó_utasítás:
`goto` *azonosító* `;`
`continue ;`
`break ;`
`return` *kifejezés*$_{opc}$ `;`

A `goto` utasításban szereplő azonosítónak az aktuális függvényben (amelyben a `goto` utasítást kiadtuk) lévő címkének kell lennie. Az utasítás hatására a vezérlés átadódik a címkézett utasításra.

A `continue` utasítás a ciklusszervező utasításokban jelenhet meg. A `continue` utasítás hatására a vezérlés átadódik a legbelső ciklus folytatását vezérlő részre (vagyis a ciklusmagot alkotó utasítás további végrehajtása lezárul és újra elindul a ciklus tesztelése). Pontosabban megfogalmazva a `continue` utasítás hatása az egyes ciklusszervező utasításokban ugyanaz, mint a `goto contin` utasításé a következő ciklusokban:

```
while (...) {        do {                  for (...) {
    ...                  ...                   ...
contin: ;            contin: ;             contin: ;
}                    } while (...) ;       }
```

A `break` utasítás csak ciklusszervező vagy `switch` utasításokban jelenhet meg. A `break` hatására befejeződik a ciklusmagot alkotó utasítás végrehajtása, a vezérlés a lezáró utasítást követő utasításra adódik.

Egy függvény a hívó eljárásba a `return` utasítás hatására tér vissza. Amikor a `return` utasítás után kifejezés áll, annak értékét a függvény visszaadja a hívó eljárásnak. A kifejezés típusa az értékadásnak megfelelően konvertálódik a függvény által meghatározott visszatérési típusra.

Ha a függvény végén nincs `return` utasítás (a vezérlés „kifolyik" a függvényből), akkor az egyenértékű egy olyan visszatéréssel, mintha nem lenne a `return` utasítás után kifejezés. Ezekben az esetekben a visszatérési érték nincs definiálva.

A10. Külső deklarációk

A C fordítóprogram számára átadott szöveges bemeneti egységet fordítási egységnek nevezzük. Egy fordítási egység külső deklarációk sorozatából áll, amelyben deklarációk vagy függvénydefiníciók lehetnek. A szintaktikai leírás:

fordítási_egység:
külső_deklaráció
fordítási_egység külső_deklaráció
külső_deklaráció:
függvénydefiníció
deklaráció

A külső deklarációk érvényességi tartománya azon fordítási egység végéig tart, amelyben deklarálva voltak, csakúgy, mint ahogy a blokkon belüli deklarációk érvényességi tartománya a blokk végéig tart. A külső deklarációk szintaxisa ugyanaz, mint az összes többi deklarációé, kivéve, hogy függvényeket csak ezen a szinten lehet deklarálni (azaz csak itt adható meg a függvényt alkotó programkód).

A10.1. Függvénydefiníciók

A függvények definíciója a következő alakban adható meg:

függvénydefiníció:
deklaráció_specifikátorok$_{opc}$ *deklarátor*
deklarációs_lista$_{opc}$ *összetett_utasítás*

A deklarációspecifikátorok közül csak az `extern` vagy `static` tárolásiosztálys-pecifikátor a megengedett, és a közöttük lévő különbségre az A11.2. pontban térünk ki.

Egy függvény visszatérési értéke aritmetikai típusú, struktúra, union, mutató vagy `void` típus lehet; de függvénnyel vagy tömbbel való visszatérés nem lehetséges. A függvénydeklarációban lévő deklarátornak explicit módon meg kell határozni, hogy a deklarált azonosító függvény típusú, vagyis az alábbi alakok egyikét tartalmaznia kell (l. az A8.6.3. pontot):

direkt_deklarátor (*paraméter_típus_lista*)
direkt_deklarátor (*azonosító_lista*$_{opc}$)

ahol a direkt deklarátor egy azonosító vagy egy zárójelezett azonosító. Főként nem szabad a függvény típust a `typedef`-fel deklarálni.

A fenti két forma közül az első az új stílusú függvénydefiníció, ami a függvény paramétereit, azok típusával együtt a paramétertípus-listában deklarálja. Ilyen esetben a deklarátorlistát követő függvénydeklarátornak hiányoznia kell. Eltekintve attól az esettől, amikor a paraméterlista csak a `void` típusjelzést tartalmazza (ami azt jelzi, hogy a függvénynek nincsenek paraméterei), a paramétertípus-lista egyes deklarátorainak azonosítót kell tartalmaznia. Ha a paraméterlista a `", ..."` karakterekkel végződik, akkor a függvény több argumentummal hívható, mint a megadott paraméterek száma. Ilyen esetekben az `<stdarg.h>` standard headerben definiált és a B. Függelékben leírt, `va_arg` makró használatán alapuló eljárással lehet a többletargumentumokhoz hozzáférni. A változó hosszúságú paraméterlistájú függvényeknek kell hogy legyen legalább egy névvel hivatkozott paramétere.

A második alak a régi stílusú definíció: az azonosító lista megnevezi a paramétereket, amíg a deklarációs lista hozzájuk rendeli a típust. Ha a paraméterekre nincs megadva deklaráció, akkor azok típusát a fordítóprogram `int`-nek tekinti. A deklarációs listának csak a listában megnevezett paramétereket kell deklarálnia, a paraméterek inicializálása nem megengedett, és csak a `register` tárolásiosztály-specifikátor használható.

Mindkét stílusú függvénydefiníció esetén a paraméterek magától értetődően a függvény magját képező összetett utasítás kezdetén deklarálódnak, és így ugyanazt az azonosítót nem szabad ott újra deklarálni (de természetesen más azonosítókhoz hasonlóan azok szintén újra deklarálhatók a külső blokkban). Ha a paraméter `T` *típusú* tömbként lett deklarálva, akkor a deklaráció „`T` *típushoz* tartozó mutató" típusúra alakul át, hasonlóan, ha a paraméter „`T` *típusú* értékkel visszatérő függvény" típusúnak lett deklarálva, akkor a deklaráció „`T` *típusú* értékkel visszatérő függvényhez tartozó mutató" típusra alakul át.

A függvény hívása során az argumentumok típusa a szükséges módon átalakul és értéke értékadással átadódik a paramétereknek.

> Az új stílusú függvénydefiníció új az ANSI szabványban. A szabvány kissé megváltoztatta a típus-előléptetés részleteit is: a könyv első kiadása még úgy specifikálta, hogy a `float` típusúnak deklarált paraméter `double` típusúvá alakul. A típus-előléptetésben meglévő kis különbségek elsősorban akkor válnak észrevehetővé, ha egy paraméterhez tartozó mutató a függvényben generálódik.

A következőkben egy teljes példát mutatunk a függvények új stílusú definíciójára.

```
int max(int a, int b, int c)
{
    int m;
    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Itt `int` a deklarációspecifikátor; `max (int a, int b, int c)` a függvény deklarátora és `{...}` a függvény programkódját tartalmazó blokk. A megfelelő régi stílusú deklaráció a következőképpen nézne ki:

```
int max(a, b, c)
int a, b, c;
{
    /* a függvény utasításai */
}
```

Itt most `int max(a, b, c)` a deklarátor és `int a, b, c;` a paraméterek deklarációs listája.

A10.2. Külső deklarációk

A külső (external) deklarációk objektumok, függvények és más azonosítók jellemzőit specifikálják. A külső megnevezés a függvényen kívüli elhelyezkedésre utal és nincs közvetlen kapcsolata az `extern` kulcsszóval. A külsőleg deklarált objektumok tárolási-osztály-meghatározása üresen hagyható, vagy `extern`, ill. `static` típusúnak adható meg.

Adott fordítási egységen belül ugyanannak az azonosítónak számos külső deklarációja létezhet, ha azok típusa és csatolása megegyezik és ha az azonosítónak létezik legfeljebb egy definíciója.

Egy objektum vagy függvény kétféle deklarációval megadott típusa akkor egyezik, ha teljesülnek az A8.10. pontban leírt szabályok. Ehhez kiegészítésül még egy szabály tartozik: ha a deklarációk azért különböznek, mert az egyik típus egy nemteljes struktúra, union vagy felsorolás (A8.3.) és a másik az ugyanolyan címkéjű, megfelelő, teljessé tett típus, akkor a típusok megegyeznek. Mi több, ha az egyik egy nemteljes tömb típus, a másik pedig egy teljes tömb típus (A8.6.2.), akkor a típusok – ha máskülönben azonosak – szintén megegyeznek. Végül, ha az egyik típus egy régi stílusú függvényt határoz meg, a másik pedig – egy különben azonos – új stílusú függvényt, a megfelelő paraméter-deklarációkkal, akkor a két típus megegyezik.

Ha egy függvény vagy objektum első külső deklarációja tartalmazza a `static` specifikátort, akkor az azonosítónak belső csatolása van, máskülönben pedig külső csatolású. A csatolás fogalmát és jellegzetességeit az A11.2. pontban tárgyaljuk.

Egy objektumra vonatkozó külső deklaráció egyben definíció is, ha a deklaráció tartalmaz kezdetiérték-adást. Ha egy külső objektum deklarációja nem tartalmaz kezdetiérték-adást és `extern`specifikátort, akkor az egy ún. próbadefiníció. Ha egy objektum definíciója megjelenik a fordítási egységben, akkor az összes próbadefiníciót a fordítóprogram redundáns deklarációként kezeli. Ha az objektum nincs definiálva a fordítási egységben, akkor az összes rá vonatkozó próbadefiníció egyetlen, nulla kezdeti értékű definícióvá válik.

Minden objektumnak egy és csak egy definíciója kell hogy legyen. A belső csatolású objektumok esetén ez a szabály minden egyes fordítási egységre külön-külön érvényes, mivel a belső csatolású objektumok egy fordítási egységre nézve egyediek. Külső csatolású objektumok esetén ez a szabály a teljes programra vonatkozik.

> Bár az egyszeri definiálás szabályát a könyv első kiadásában némiképp különböző formában fogalmaztuk meg, a hatás azonos az itt megfogalmazottal. Néhány gépi megvalósítás enyhít ezen a szabályon a próbadefiníció fogalmának általánosításával. Egy, a UNIX rendszerben szokásos és a szabvány általánosan használt kiterjesztéseként felfogható alternatíva szerint egy külső csatolású objektum összes próbadefinícióit a programot alkotó összes fordítási egységre együttesen kezelik, az itt megfogalmazott, fordítási egységenkénti kezelés helyett. Ilyen esetben, ha egy definíció bárhol a programban előfordul, akkor a próbadefiníciók puszta deklarációkká válnak, ha viszont nem jelenik meg a definíció, akkor az összes próbadefiníció nulla kezdeti értékű definícióvá válik.

A11. Érvényességi tartomány és csatolás

Egy C nyelvű programot nem szükséges egyszerre lefordítani: a forrásprogram több, fordítási egységeket tartalmazó állományban tartható és az előre lefordított eljárások a könyvtárakból tölthetők be. A programot alkotó függvények közötti kommunikáció a függvényhívásokon és a külső adatok felhasználásán keresztül megy végbe.

Ezért kétféle érvényességi tartományt kell megkülönböztetnünk: egy azonosító lexikális érvényességi tartományát, ami a program szövegének az a része, ahol az azonosító ismert és tulajdonságai meghatározottak, valamint a külső csatolású objektumokkal és függvényekkel kapcsolatos érvényességi tartományt, amely meghatározza a különálló fordítási egységekben lévő azonosítók közötti kapcsolatot.

A11.1. Lexikális érvényességi tartomány

Az azonosítók számos névtér valamelyikébe sorolhatók és ezek a névterek egymástól függetlenek, az oda tartozó azonosítók nincsenek hatással a másik névtér azonosítóira. Ugyanaz az azonosító különböző célra, azonos érvényességi tartománnyal használható, ha különböző névtérbe tartozik. A C nyelvben többféle névtér létezik: objektumok, függvények, `typedef` nevek, ill. felsorolt állandók; címkék; struktúrák, unionok és felsorolások címkéi; önállóan a struktúrák és unionok tagjai.

> Ezek a szabályok többféle módon is különböznek a könyv első kiadásában közöltektől. A címkéknek az előzőekben nem volt saját névterük, a struktúrák és unionok címkéi önálló névteret alkottak és néhány gépi megvalósításban a felsorolások címkéi is. A különböző címkék azonos névtérbe helyezése új korlátozást jelent. A legfontosabb eltérés az előző kiadáshoz képest, hogy az egyes struktúrák és unionok a saját tagjaik számára önálló névteret képeznek, így ugyanaz a név megjelenhet különböző struktúrákban vagy unionokban. Ezt a szabályt már néhány éve általánosan alkalmazzák.

Egy objektum vagy függvény azonosítójának lexikális érvényességi tartománya egy külső deklarációban a deklarátor befejezésével (lezárásával) kezdődik és a deklarációt tartalmazó fordítási egység végéig tart. Egy függvénydefinícióban szereplő paraméter érvényességi tartománya a függvénydefiniálás blokkjának kezdetén indul és végig a függvényben érvényes; egy függvénydeklarációban szereplő paraméter érvényességi tartománya viszont csak a deklarátor végéig terjed. Egy blokk fejrészében deklarált azonosító érvényességi tartománya a deklarátor végétől (lezárásától) a blokk végéig terjed. Egy címke érvényességi tartománya a teljes függvény, amelyben megjelenik. Egy struktúra, union vagy felsorolás címkéjének vagy egy felsorolt állandónak az érvényességi tartománya a típusspecifikátorban való megjelenésével kezdődik és a fordítási egység végéig (külső szintű deklaráció esetén) vagy a blokk végéig (függvényen belüli deklaráció esetén) tart.

Ha egy azonosítót explicit módon deklarálunk egy blokk fejrészében, beleértve a függvényt alkotó blokkot is, akkor az azonosító bármilyen blokkon kívüli deklarációja a blokk végéig fel lesz függesztve.

A11.2. Csatolás

Egy fordítási egységen belül ugyanazon belső csatolású objektum vagy függvény azonosítójának minden deklarációja ugyanazt a dolgot jelenti, és az objektum vagy függvény a fordítási egységre vonatkoztatva egyedi. Ugyanazon objektum vagy függvény külső csatolású azonosítójára vonatkozó összes deklaráció szintén ugyanazt a dolgot jelenti, és az objektum vagy függvény a teljes programban, bármely eljárás számára használható.

Ahogyan ezt már az A10.2. pontban elmondtuk, egy azonsoító első külső deklarációja `static` tárolásiosztály-specifikátor alkalmazása esetén belső csatolást, minden más specifikátor esetén pedig külső csatolást eredményez. Ha egy azonosító blokkon belüli deklarációja nem tartalmazza az `extern` tárolásiosztálys-pecifikátort, akkor az azonosítónak nincs csatolása és a függvényre vonatkoztatva egyedi. Ha a deklaráció tartalmazza az `extern` tárolásiosztály-specifikátort és egy, az azonosítóra vonatkozó külső deklaráció aktív a környező blokkra vett érvényességi tartományban, akkor az azonosító ugyanolyan csatolású, mint a külső deklaráció, és ugyanazt az objektumot vagy függvényt jelenti. Ha az érvényességi tartományban nincs külső deklaráció, akkor a csatolás külső.

A12. Az előfeldolgozó rendszer

Az előfeldolgozó rendszer feladata a makrók kifejtése és behelyettesítése, a feltételes fordítás vezérlése, valamint a megnevezett állományok programba építése. A forrásprogram # jellel kezdődő (előtte üreshely-karakterek használhatók) sorai az előfeldolgozó rendszernek szóló információkat tartalmaznak. Ezeknek a soroknak a szintaxisa független a C nyelv többi részétől, a programban bárhol megjelenhetnek és hatásuk (az érvényességi tartománytól függetlenül) a fordítási egység végéig tart. A sorokra tördelés lényeges, mivel az előfeldolgozó rendszer minden sort egyedileg analizál (bár több sor is összekapcsolható az A12.2. pontban leírtak szerint). Az előfeldolgozó rendszer számára bármilyen nyelvi szintaktikai egység (token) egy önálló szintaktikai egységet alkot és egy karaktersorozat egy állománynevet ad meg (mint pl. az `include` direktívában, l. az A12.4. pontot). A feldolgozás során minden karakter, ami nincs másképpen definiálva, szintén szintaktikai egységet képez. Az előfeldolgozó rendszernek szóló sorokban lévő, szóköztől és a vízszintes tabulátortól különböző üreshely-karakterek jelentése (hatása) nincs definiálva.

Az előfeldolgozás több, egymástól elkülönülő, logikailag egymásra épülő fázisból áll, amelyek a konkrét gépi megvalósításban összevonhatók. A feldolgozás fázisai:

1. Az előfeldolgozó rendszer az A12.1. pontban leírt trigráf karaktersorozatokat helyettesíti a megfelelő karakterekkel. Amennyiben az operációs rendszer megköveteli, akkor újsor-karakterek épülnek be a forrásállomány egyes sorai közé.
2. Minden egyes backslash karakterből (\) és az azt követő újsor-karakterből álló kombináció törlődik, amivel az összetartozó sorok egy sorrá egyesülnek (A12.2.).
3. A program felbontása üreshely-karakterekkel elválasztott szintaktikai egységekre (tokenekre); a magyarázó szövegek helyettesítése egyetlen szóközzel. Ekkor az előfeldolgozást vezérlő direktívák végrehajtódnak és megtörténik a makrók (A12.3. – A.12.10.) kifejtése.
4. A karakteres és karaktersorozat-állandókban lévő escape-sorozatokat a rendszer helyettesíti a megfelelő karakterekkel (A2.5.2., A2.6.) és a szomszédos karaktersorozat-állandók konkatenálódnak.
5. Az így előkészített program fordítása, ill. más programokkal és könyvtárakkal való összeszerkesztése. Ez a szükséges programok és adatok kigyűjtése, valamint a külső függvényekre és objektumokra való hivatkozások definíciójukkal történő összekapcsolása alapján megy végbe.

A12.1. Trigráf karaktersorozatok

A C nyelvű forrásprogramok a hétbites ASCII karakterkészletet használják, de létezik az ISO 646-1983 Invariáns Kódrendszer bővített változata is. Azért, hogy a programok a redukált karakterkészlettel is ábrázolhatók legyenek, a következő trigráf karakter sorozatokat a megfelelő karakterrel kell helyettesíteni:

```
??= #          ??( [          ??< {
??/ \          ??) ]          ??> }
??' ^          ??! |          ??- ~
```

A helyettesítés minden más feldolgozási lépést megelőzően történik, és más ilyen helyettesítés nem fordul elő.

A trigráf karaktersorozatok bevezetése új az ANSI szabványban.

A12.2. Sorok egyesítése

A sor végén elhelyezett backslash karakter (\) azt jelzi, hogy az előfeldolgozó rendszernek szánt információ a következő sorban folytatódik. Az előfeldolgozó rendszer a program szintaktikai egységekre bontása előtt a \ karakter és az azt követő újsor-karakter törlésével ezeket a sorokat egyesíti.

A12.3. Makrók definíciója és kifejtése

A programban előforduló

```
#define azonosító token_sorozat
```

alakú vezérlősor arra utasítja az előfeldolgozót, hogy az azonosító következő előfordulását helyettesítse a szintaktikai egységek (tokenek) megadott sorozatával. A szintaktikai egységeket megelőző és záró üreshely-karakterek törlődnek. Ugyanerre az azonosítóra vonatkozó második `#define` sor hibajelzést eredményez, kivéve, ha a másodiknak megadott szintaktikai egység sorozat megegyezik az elsővel, az összes üreshely-karakterből álló elválasztásokat is figyelembe véve.

A

`#define` *azonosító*`(` *azonosító_lista* `)` *token_sorozat*

alakú sor (ahol az első azonosító és a kezdő kerek zárójel között nincs szóköz!) egy makrót definiálnak az azonosítólistában megadott paraméterekkel. Csakúgy, mint az első alak esetén, a szintaktikai egységek előtt és után lévő üreshely-karakterek törlődnek, és a makrót csak úgy lehet újradefiniálni, ha az új és régi definícióban a paraméterek száma és leírása, valamint a tokenek sorozata megegyezik.

Az

`#undef` *azonosító*

alakú vezérlősor hatására az előfeldolgozó rendszer „elfelejti" az azonosító korábbi, `#define` sorral megadott definícióját. Egy ismeretlen azonosítóra kiadott `#undef` direktíva nem jelent hibát.

Amikor a másodiknak megadott forma szerint definiálunk egy makrót, akkor annak hívása a makró azonosítójából, az azt követő kerek nyitó zárójelből, szintaktikai egységek vesszővel elválasztott sorozatából és egy kerek végzárójelből áll. (Természetesen üreshely-karakterek megengedettek.) A hívás argumentumait szintaktikai egységek vesszővel elválasztott sorozata alkotja, a szövegben lévő vesszőket aposztrófok vagy zárójelek között kell elhelyezni, hogy ne elválasztó jelként hassanak. Az előfeldolgozó rendszer információgyűjtési fázisában az argumentumokra még nincs makrókifejtés. Híváskor az argumentumok számának meg kell egyezni a definícióban lévő paraméterek számával. Az argumentumok szétválasztása után az előfeldolgozó rendszer eltávolítja a bevezető és záró üreshely-karaktereket. Mindezek után az előfeldolgozó az egyes argumentumokból így előállított token-sorozattal helyettesíti a megfelelő paraméterazonosító minden egyes nem aposztrófok közötti előfordulását a makró helyettesítő token-sorozatában. Hacsak a helyettesítősorozatban a paramétert nem # előzi meg, vagy ## van előtte és utána, akkor az argumentum tokeneket a makróhívás szempontjából újra megvizsgálja, és ha szükséges, akkor a helyettesítő szöveg beiktatása előtt megtörténik a makrókifejtés.

A helyettesítési folyamatot két speciális operátor befolyásolja. Az első esetben, ha a helyettesítő token-sorozatban egy paraméter előfordulását közvetlenül a # előzi meg, akkor a megfelelő paraméter elé és után az idézőjel-karakter kerül, és mind a `#`, mind a paraméter-azonosító helyettesítődik az idézőjelek közötti argumentummal. Minden egyes idézőjel vagy az argumentumban lévő karakteres állandóban, ill. karaktersorozat-állandóban előforduló \ karakter elé egy \ karakter iktatódik.

A második esetben, ha valamelyik makró definíciója egy ## operátort tartalmaz, akkor a paraméter helyettesítése után az egyes ## jelek törlődnek (együtt annak két oldalán lévő üreshely-karakterekkel), aminek hatására a behelyettesített szöveg konkatenálódik a szomszédos tokennel és egy új tokent eredményez. Ha ezzel a módszerrel érvénytelen token jön létre, ill. ha az eredmény függ a ## operátorok feldolgozási sorrendjétől, akkor a hatás definiálatlan. A ## nem jelenhet meg egy helyettesítő token-sorozat kezdetén vagy végén.

Mindkét fajta makró feldolgozásakor a rendszer a helyettesített token-sorozatot ismételten átvizsgálja, hogy megtalálja az előfeldolgozó az esetleges további definiálatlan azonosítókat. Mindenesetre, ha egyszer egy azonosító egy adott kifejezésben helyettesítődött, akkor az nem helyettesítődik újra az újbóli átvizsgálás során, hanem változatlan marad.

Ha egy makrókifejtés végső értéke # jellel kezdődik, akkor azt az előfeldolgozó rendszer már nem tekinti direktívának.

> A makrókifejtési folyamat részleteit az ANSI szabvány sokkal pontosabban írja le, mint a könyv első kiadása. A legfontosabb változás a korábbiakhoz képest a # és ## operátorok bevezetése, amelyekkel megoldható szövegek idézőjelek közé tétele, ill. konkatenálása. A szabvány néhány új szabálya, különösen a konkatenációra vonatkozóak, kissé bizarr (l. az alábbi példát).

Például ez a lehetőség kihasználható a „manifesztálódott" állandók létrehozásánál, mint

```
#define TABMERET 100
int tabla[TABMERET];
```

esetén. A

```
#define ABSDIFF(a, b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

definíció egy makrót definiál, amely visszatér az argumentumai közötti különbség abszolút értékével. Eltérően egy ugyanezen feladatot ellátó függvénytől, itt az argumentumok és a visszatérési érték tetszőleges aritmetikai típusú vagy mutató lehet. További előnye a makrónak, hogy az argumentumok, amelyek mellékhatásokat okozhatnak, kétszer lesznek kiértékelve: egyszer az ellenőrzéshez és másodszor az érték előállításához.

A

```
#define tempfile(dir) #dir "/%s"
```

definíció a `tempfile(/usr/tmp)` makróhívást eredményezi, amelyben az

```
"/usr/tmp" "/%s"
```

egymást követő karaktersorozatok egyetlen karaktersorozattá olvadnak össze (konkatenálódnak). A

```
#define cat(x, y) x ## y
```

definíció után a `cat(var, 123)` makróhívásban az argumentum `var123` lesz. Viszont a `cat(cat(1,2), 3)` makróhívás definiálatlan eredményt ad, mivel a ## operátor jelenléte megvédi a külső makróhívás argumentumait a makrókifejtéstől. Így ez az előfeldolgozó rendszerrel feldolgozva a

```
cat ( 1 , 2 )3
```

token karaktersorozatot eredményezi, és ebben a `)3` (ami az első argumentum utolsó és a második argumentum első tokenjének összekapcsolódása) nem legális token. Ha bevezetjük a

```
#define xcat(x, y) cat(x, z)
```

második szintű makródefiníciót, akkor a helyzet egyszerű lesz, az `xcat(xcat(1, 2), 3)` már valóban az `123` eredményt adja, mivel az `xcat` kifejtése nem tartalmazza a `##` operátort.

Hasonló módon az `ABSDIFF(ABSDIFF(a, b), c)` a várt, teljesen kifejtett eredmény adja.

A12.4. Állományok beépítése

A

```
#include <állománynév>
```

alakú vezérlősor hatására a sor helyettesítődik a megadott nevű állomány teljes tartalmával. Az állománynévben nem szerepelhet a > vagy az újsor-karakter, és ha a `",  ',\` vagy `/*` szerepel a névben, akkor az eredmény definiálhatatlan. A megadott nevű állományt a rendszer a gépi megvalósítástól függő helyen, soros módon fogja keresni.

A

```
#include "állománynév"
```

vezérlősor ugyanúgy működik, mint az előző alak, csak az állomány keresése először az eredeti forrásállomány helyén indul, és ha ott sikertelen, akkor folytatódik az első alaknak megfelelő módon. Az állománynévben lévő `', \` vagy `/*` karakterek hatása itt is definiálatlan, viszont a > karakter szerepelhet az állománynévben.

Végül a

```
#include token_sorozat
```

alakú direktíva hatására a rendszer a token sorozatot normális szövegkénti kifejtés után értelmezi. A kifejtésnek <...> vagy "..." alakot kell eredményezni, és a hatás a kapott eredménynek megfelelő lesz.

A `#include` direktívák egymásba ágyazhatók (tehát a beépített állomány tartalmazhat további `#include` direktívákat).

A12.5. Feltételes fordítás

Egy program meghatározott része feltételtől függően fordítható le az alábbi vázlatos szintaktikai leírás szerint:

előfeldolgozó_feltételes_fordítás:
    if_sor szöveg elif_rész else_rész$_{opc}$ `#endif`

if_sor:
    `#` `if` *állandó_kifejezés*
    `#` `ifdef` *azonosító*
    `#` `ifndef` *azonosító*

elif_rész:
    elif_sor szöveg

elif_rész$_{opc}$

elif_sor
    `#elif` *állandó_kifejezés*

else_rész:
    else_sor szöveg
    else_sor:
    `#else`

Az egyes direktívák (if–sor, elif–sor, else–sor és `#endif`) egy sorban, önállóan jelennek meg. Az `#if` direktívában és az azt követő `#elif` direktívákban lévő állandó kifejezéseket a program sorban egymás után addig értékeli ki, amíg nem nulla értékű kifejezést talál. Nulla érték esetén a sorban következő szöveget törli, sikeres (nem nulla) esetben pedig normális módon feldolgozza. Szöveg alatt olyan tetszőleges információ – beleértve az előfeldolgozónak szánt direktívákat is – értendő, ami nem része a feltételes szerkezetnek. A szöveg rész üres is lehet. Ha az előfeldolgozó rendszer egy sikeres (nem nulla értékű kifejezést tartalmazó) `#if` vagy `#elif` sort talált és a szöveget feldolgozta, akkor a további `#elif` és `#else` sorokat, együtt a bennük elhelyezett szöveggel törli. Ha minden kifejezés nulla értékű és létezik `#else`, akkor az `#else` utáni szöveget dolgozza fel a szokásos módon. A feltétel inaktív részével vezérelt szöveg a feltételek beágyazásának ellenőrzését kivéve törlődik.

Az `#if` és `#elif` után álló állandó kifejezést az előfeldolgozó rendszer egy közönséges makróhelyettesítési menettel dolgozza fel. Az előfeldolgozó a

`defined` *azonosító*

vagy

`defined (` *azonosító* `)`

alakú kifejezéseket a makrókra való ellenőrzés előtt `1L` értékkel helyettesíti, ha az azonosító már definiálva van a számára és `0L` értékkel, ha még nincs. A makrókifejtés után fennmaradó azonosítók (amelyek nem lettek definiálva) a `0L` értékkel helyettesítődnek. A rendszer minden egyes egész típusú állandót az `L` utótaggal egészíti ki, így az aritmetika `long` vagy `unsigned long` típusú.

A direktívákban szereplő állandó kifejezés (A7.19.) egész típusú kell hogy legyen és nem szerepelhet benne `sizeof`, kényszerített típuskonverzió, valamint felsorolt állandó. Az

`#ifdef` *azonosító*
`#ifndef` *azonosító*

alakú vezérlősorok egyenértékűek az
`#if defined` *azonosító*
`#if ! defined` *azonosító*

alakú vezérlősorokkal.

> Az `#elif` a könyv első kiadása óta megjelent új direktíva, bár néhány előfeldolgozó rendszer már korábban is használta. Az előfeldolgozó rendzser `defined` operátora szintén új.

A12.6. Sorvezérlés

A C nyelvű programokat létrehozó előfeldolgozó rendszerek számára hasznosak a

```
#line állandó "állománynév"
#line állandó
```

alakú vezérlősorok, amelyek hatására a fordítóprogram azt hiszi, hogy a következő forrássor sorszáma a decimális egész állandóval megadott érték és az aktuális bemeneti állomány az, amelynek a nevét az azonosítóval megadtuk. Ha az idézőjelek közötti állománynév hiányzik, akkor a korábban megjegyzett állománynév marad érvényben. A vezérlősorokban elhelyezett makrókat a sor értelmezése előtt kifejti a rendszer. Az ilyen vezérlősorok főképp diagnosztikai célra használhatók.

A12.7. Hibaüzenet generálása

Az előfeldolgozó rendszernek kiadott

```
#error token_sorozat(opc)
```

alakú vezérlősor hatására az előfeldolgozó rendszer a token sorozatot magában foglaló hibaüzenetet ír ki.

A12.8. A pragma direktíva

Az előfeldolgozó rendszernek kiadott

```
#pragma token_sorozat(opc)
```

alakú vezérlősor hatására egy, a gépi megvalósítástól függő hatás jön létre. Az előfeldolgozó rendszer a fel nem ismert pragma direktívákat figyelmen kívül hagyja.

A12.9. A nulldirektíva

Az előfeldolgozó rendszernek kiadott

```
#
```

alakú vezérlősor hatására nem történik semmi (nulldirektíva).

A12.10. Előre definiált nevek

Néhány azonosító az előfeldolgozó rendszer számára előre definiálva van és kifejtésükkel speciális információ állítható elő. Ezek, valamint az előfeldolgozó hozzájuk tartozó kifejezésoperátorai `defined` típusúak, nem lehetnek definiálatlanok és nem definálhatók újra. Az előre definiált nevek és jelentésük:

__LINE__ A forrásprogram éppen feldolgozás alatt álló aktuális sorának sorszámát tartalmazó decimális állandó.

__FILE__ Az éppen fordítás alatt álló forrásállomány nevét tartalmazó karaktersorozat-állandó.

__DATE__ A fordítás dátumát "Hhh nn éééé" alakban tartalmazó karaktersorozat-állandó.

__TIME__ A fordítás időpontját "óó:pp:ss" alakban tartalmazó karaktersorozat-állandó.

__STDC__ Állandó 1 érték. Ezzel az azonosítóval az volt a szándék, hogy az 1-nek definiált értékkel jelezze a szabványhoz illeszkedő gépi megvalósítást.

Az #error és #pragma direktívák újak az ANSI szabványban. Az előre definiált előfeldolgozó makrók szintén újak, bár néhányat közülük az egyes gépi megvalósítások már korábban is használtak.

A13. A C nyelv szintaktikájának összefoglalása

A következőkben összefoglaljuk a C nyelv korábbi részekben megadott szintaktikáját. Ez a leírás tartalmilag pontosan egyezik az A. Függelékben leírtakkal, csak a sorrendje más.

A szintaktikai leírás nem definiálja az egész állandó, karakteres állandó, lebegőpontos állandó, azonosító, karaktersorozat- és felsorolt állandó terminális szimbólumokat, és a szövegben a programokhoz használt betűtípussal szedett szavak és szimbólumok szintén terminálisak. Az itt megadott szintaktikai leírás mechanikusan átalakítható egy automatikus szintaktikai elemző rendszerré. A leírás általában soronként tagolva adja meg az egyes lehetőségeket, de néhány helyen (helytakarékosságból) szükség volt az „egyike a következőknek:" szerkezetre, valamint az egyes lehetőségek megduplázására *opc* kiegészítéssel és kiegészítés nélkül. Még egy változás van a korábbiakhoz képest: a leírásból kihagytuk a *typedef_név:azonosító* szintaktikai egységet és helyette a *typedef_név* terminális szimbólumot használjuk, amivel a szintaktikai leírás a YACC elemző számára is elfogadhatóvá vált, egyetlen konfliktushelyzetet, az `if-else` kétértelműséget leszámítva.

A C nyelv szintaktikája:

fordítási_egység:
    külső_deklaráció
    fordítási_egység külső_deklaráció

külső_deklaráció:
    függvénydefiníció
    deklaráció

függvénydefiníció:
    deklaráció_specifikátorok$_{opc}$ *deklarátor deklarációs_lista*$_{opc}$ *összetett_utasítás*

deklaráció:
    deklaráció_specifikátorok kezdeti_deklarátor_lista$_{opc}$;

deklarációs_lista:
    deklaráció
    deklarációs_lista deklaráció

deklaráció_specifikátorok:
tárolási_osztály_specifikátor deklaráció_specifikátorok$_{opc}$

típus_specifikátor deklaráció_specifikátorok$_{opc}$
típus_minősítő deklaráció_specifikátorok$_{opc}$

tárolási_osztály_specifikátor: egyike a következőknek:
`auto register static extern typedef`

típus_specifikátor: egyike a következőknek:
`void char short int long float double signed unsigned`
struktúra_vagy_union_specifikátor felsorolás_specifikátor typedef_név

típus_minősítő: egyike a következőknek:
`const volatile`

struktúra_vagy_union_specifikátor:
struktúra_vagy_union azonosító$_{opc}$ { *struktúra_deklarációs_lista* }
struktúra_vagy_union azonosító

struktúra_vagy_union: egyike a következőknek:
`struct union`

struktúra_deklarációs_lista:
struktúra_deklaráció
struktúra_deklarációs_lista struktúra_deklaráció

kezdeti_deklarátor_lista:
kezdeti_deklarátor
kezdeti_deklarátor_lista , kezdeti_deklarátor

kezdeti_deklarátor:
deklarátor
deklarátor = kezdeti_érték

struktúra_deklaráció:
specifikátor_minősítő_lista struktúra_deklarátor_lista ;

specifikátor_minősítő_lista:
típus_specifikátor specifikátor_minősítő_lista$_{opc}$
típus_minősítő specifikátor_minősítő_lista$_{opc}$

struktúra_deklarátor_lista:
struktúra_deklarátor
struktúra_deklarátor_lista , struktúra_deklarátor
struktúra_deklarátor:
deklarátor
deklarátor$_{opc}$: *állandó_kifejezés*

felsorolás_sepcifikátor:
enum *azonosító*$_{opc}$ { *felsorolás_lista* }
enum *azonosító*

felsorolás_lista:
felsorolt_elem
felsorolás_lista , *felsorolt_elem*

felsorolt_elem:
azonosító
azonosító = *állandó_kifejezés*

deklarátor:
mutató$_{opc}$ *direkt_deklarátor*

direkt_deklarátor:
azonosító
{ *deklarátor* }
direkt_deklarátor [*állandó_kifejezés*$_{opc}$]
direkt_deklarátor (*paraméter_típus_lista*)
direkt_deklarátor (*azonosító_lista*$_{opc}$)

mutató:
* *típus_minősítő_lista*$_{opc}$
* *típus_minősítő_lista*$_{opc}$ *mutató*

típus_minősítő_lista:
típus_minősítő
típus_minősítő_lista típus_minősítő

paraméter_típus_lista:
paraméter_lista
paraméter_lista , . . .

paraméter_lista:
paraméter_deklaráció
paraméter_lista , *paraméter_deklaráció*

paraméter_deklaráció:
deklaráció_specifikátorok deklarátor
deklaráció_specifikátorok absztrakt_deklarátor$_{opc}$
azonosító_lista:
azonosító
azonosító_lista , *azonosító*
kezdeti érték:
értékadó_kifejezés
{ *kezdetiérték-lista* }
{ *kezdetiérték-lista* , }

kezdetiérték-lista:
    kezdeti érték
    kezdetiérték-lista , *kezdeti érték*

típusnév:
    specifikátor_minősítő_lista absztrakt_deklarátor$_{opc}$

absztrakt_deklarátor:
    mutató
    mutató$_{opc}$ *direkt_absztrakt_deklarátor*

direkt_absztrakt_deklarátor:
    { *absztrakt_deklarátor* }
    direkt_absztrakt_deklarátor$_{opc}$ [*állandó_kifejezés*$_{opc}$]
    direkt_absztrakt_deklarátor$_{opc}$ (*paraméter_típus_lista*$_{opc}$)

typedef_név:
    azonosító

utasítás:
    címkézett_utasítás
    kifejezés_utasítás
    összetett_utasítás
    kiválasztó_utasítás
    iterációs_utasítás
    vezérlésátadó_utasítás

címkézett_utasítás:
    azonosító : *utasítás*
    `case` *állandó_kifejezés* : *utasítás*
    `default` : *utasítás*

kifejezésutasítás:
    kifejezés$_{opc}$;

összetett_utasítás:
    { *deklarációs_lista*$_{opc}$ *utasítás_lista*$_{opc}$ }

utasítás_lista:
    utasítás
    utasítás_lista utasítás

kiválasztó_utasítás:
    `if` (*kifejezés*) *utasítás*
    `if` (*kifejezés*) *utasítás* else *utasítás*
    `switch` (*kifejezés*) *utasítás*
iterációs_utasítás:
    `while` (*kifejezés*) *utasítás*
    `do` *utasítás* `while` (*kifejezés*) ;
    `for` (*kifejezés*$_{opc}$; *kifejezés*$_{opc}$; *kifejezés*$_{opc}$) *utasítás*

vezérlésátadó_utasítás:
    `goto` *azonosító* `;`
    `continue ;`
    `break ;`
    `return` *kifejezés*$_{opc}$ `;`

kifejezés:
    értékadó_kifejezés
    kifejezés `,` *értékadó_kifejezés*

értékadó_kifejezés:
    feltételes_kifejezés
    unáris_kifejezés értékadó_operátor értékadó_kifejezés

értékadó_operátor: egyike a következőknek:
    `= *= /= %= += -= <<= >>= &= ^= |=`

feltételes_kifejezés:
    logikai_VAGY_kifejezés
    logikai_VAGY_kifejezés `?` *kifejezés* `:` *feltételes kifejezés*

állandó_kifejezés:
    feltételes_kifejezés

logikai_VAGY_kifejezés:
    logikai_ÉS_kifejezés
    logikai_VAGY_kifejezés `||` *logikai_ÉS_kifejezés*

logikai_ÉS_kifejezés:
    inkluzív_VAGY_kifejezés
    logikai_ÉS_kifejezés `&&` *inkluzív_VAGY_kifejezés*

inkluzív_VAGY_kifejezés:
    kizáró_VAGY_kifejezés
    inkluzív_VAGY_kifejezés `|` *kizáró_VAGY_kifejezés*

kizáró_VAGY_kifejezés:
    ÉS_kifejezés
    kizáró_VAGY_kifejezés `^` *ÉS_kifejezés*

ÉS_kifejezés:
    egyenlőség_kifejezés
    ÉS_kifejezés `&` *egyenlőség_kifejezés*

egyenlőség_kifejezés:
    relációs_kifejezés
    egyenlőség_kifejezés `==` *relációs_kifejezés*
    egyenlőség_kifejezés `!=` *relációs_kifejezés*

relációs_kifejezés:
    léptető_kifejezés
    relációs_kifejezés < *léptető_kifejezés*
    relációs_kifejezés > *léptető_kifejezés*
    relációs_kifejezés <= *léptető_kifejezés*
    relációs_kifejezés >= *léptető_kifejezés*

léptető_kifejezés:
    additív_kifejezés
    léptető_kifejezés << *additív_kifejezés*
    léptető_kifejezés >> *additív_kifejezés*

additív_kifejezés
    multiplikatív_kifejezés
    additív_kifejezés + *multiplikatív_kifejezés*
    additív_kifejezés – *multiplikatív_kifejezés*

multiplikatív_kifejezés:
    kényszerített_típuskonverziójú_kifejezés
    multiplikatív_kifejezés * *kényszerített_típuskonverziójú_kifejezés*
    multiplikatív_kifejezés / *kényszerített_típuskonverziójú_kifejezés*
    multiplikatív_kifejezés % *kényszerített_típuskonverziójú_kifejezés*

kényszerített_típuskonverziójú_kifejezés:
    unáris_kifejezés:
    (*típusnév*) *kényszerített_típuskonverziójú_kifejezés*

unáris_kifejezés:
    utótagos_kifejezés
    ++ *unáris_kifejezés*
    -- *unáris_kifejezés*
    unáris_operátor kényszerített_típuskonverziójú_kifejezés

    `sizeof` *unáris_kifejezés*
    `sizeof (` *típusnév*)

unáris_operátor: egyike a következőknek:
    `& * + - ~ !`

utótagos_kifejezés:
    elsődleges_kifejezés
    utótagos_kifejezés [*kifejezés*]
    utótagos_kifejezés (*argumentum_kifejezés_lista*$_{opc}$)
    utótagos_kifejezés . *azonosító*
    utótagos_kifejezés –> *azonosító*
    utótagos_kifejezés ++
    utótagos_kifejezés --

elsődleges_kifejezés:
azonosító
állandó
karaktersorozat
(*kifejezés*)

argumentum_kifejezés_lista:
értékadó_kifejezés
argumentum_kifejezés_lista , *értékadó_kifejezés*

állandó:
egész_állandó
karakteres_állandó
lebegőpontos_állandó
felsorolt_állandó

A következőkben az előfeldolgozó rendszer szintaktikai leírását összegezzük. A leírás megadja a vezérlősorok szerkezetét, de nem alkalmas mechanikus elemzésre. A leírásban használjuk a szöveg szimbólumot, ami tetszőleges forrásprogram szöveget, az előfeldolgozó rendszernek szóló nem feltételes vezérlősorokat vagy az előfeldolgozónak szóló teljes feltételes konstrukciókat jelent.

Az előfeldolgozó rendszer szintaktikája:

vezérlősorok:
`#` `define` *azonosító token_sorozat*
`#` `define` *azonosító* (*azonosító* , `...` , *azonosító*) *token_sorozat*
`#` `undef` *azonosító*
`#` `include` <*állománynév*>
`#` `include` *"állománynév"*
`#` `include` *token_sorozat*
`#` `line` *állandó "állománynév"*
`#` `line` *állandó*
`#` `error` *token_sorozat*$_{opc}$
`#` `pragma` *token_sorozat*$_{opc}$
`#`
előfeldolgozó_feltételes_fordítás

előfeldolgozó_feltételes_fordítás:
if_sor szöveg elif_részek else_rész$_{opc}$ `#` `endif`

if_sor:
`#` `if` *állandó_kifejezés*
`#` `ifdef` *azonosító*
`#` `ifndef` *azonosító*

elif_részek:
elif_sor szöveg
elif_részek$_{opc}$

elif_sor:
    `# elif` *állandó_kifejezés*

else_rész:
    else_sor szöveg

else_sor:
    `# else`

A standard könyvtár

Ebben a függelékben összefoglaljuk az ANSI szabványban definiált programkönyvtárral kapcsolatos ismereteket. A standard könyvtár nem része a szűkebb értelemben vett C nyelvnek, de gondoskodik függvények deklarálásáról, valamint adattípusok és makrók definiálásáról, amivel a szabványos C nyelvet támogató környezetet hoz létre. Az ismertetésből néhány, csak korlátozottan használható vagy más függvényekből egyszerűen előállítható függvény leírását elhagytuk, csakúgy, mint a több-bájtos karakterekkel végezhető műveleteket, valamint a helyi jellegeztességektől (pl. nyelvtől) függő részleteket.

A standard könyvtár függvényei, típusai és makrói standard headerekben vannak deklarálva. Ezek a standard headerek:

```
<assert.h>    <limits.h>    <signal.h>    <stdlib.h>
<ctype.h>     <locale.h>    <stdarg.h>    <string.h>
<errno.h>     <math.h>      <stddef.h>    <time.h>
<float.h>     <setjmp.h>    <stdio.h>
```

Egy headerhez az

```
#include <headernév>
```

direktívával férhetünk hozzá. A headerek bármilyen sorrendben és akárhányszor beépíthetők a programba. Egy headert bármilyen külső deklaráción vagy definíción kívül, vagy bármely, a headerben deklarált függvény, típus vagy adat felhasználása előtt kell beépíteni. A headernek nem szükséges a forrásállományban lenni.

Az aláhúzással kezdődő külső azonosítók a könyvtár számára vannak fenntartva, és a könyvtárban használt összes azonosító aláhúzással kezdődik vagy nagybetűs írásmóddal és kezdő aláhúzással van jelölve.

B1. Adatbevitel és adatkivitel: az <stdio.h> header

Az adatbevitelt és -kivitelt kezelő függvények és makrók, valamint a hozzájuk tartozó adattípusok az `<stdio.h>` headerben vannak definiálva, és ez a header alkotja a standard könyvtár közel egyharmadát.

Az adatok forrása vagy címzettje (ami lemezegység vagy más periféria lehet) adatáramot kezel. A standard könyvtár a szöveges és a bináris adatáram használatát támogatja, bár ezek néhány operációs rendszerben (pl. a UNIX esetén is) azonosak. Egy szöveges adatáram egymást követő sorokból áll, az egyes sorok nulla vagy több karaktert tartlamaznak és `'\n'` (újsor-) karakterrel záródnak. Egy programozási környezetben szükség lehet egy szöveges adatáram valamilyen más ábrázolásmódba való alakítására (pl. a `'\n'` kocsivissza- és soremelés-karakterekké alakítására). Egy bináris adatáram

belső adatokat tartalmazó feldolgozatlan bájtok sorozata, és fő jellegzetessége, hogy kiírása után azonnal visszaolvassa a rendszer ellenőrzés céljából.

Egy adatáram a megnyitási folyamattal kapcsolódik egy állományhoz vagy perifériához és ez a kapcsolat az adatáram lezárásával szakítható meg. Az adatállományt megnyitó eljárás `FILE` típusú objektumot címző mutatóval tér vissza, és ez az objektum tartalmazza az adatáram vezérléséhez szükséges információkat. A későbbiekben, amikor nem okoz félreértést, az állománymutató és az adatáram fogalmát csereszabatosan fogjuk használni.

Amikor egy program végrehajtása megkezdődik, három adatáram, az `stdin`, `stdout` és `stderr` automatikusan megnyílik.

B1.1. Állománykezelő műveletek

A következőkben ismertetendő függvények végzik az állománykezelő műveleteket. A függvényekben használt `size_t` a `sizeof` operátorral létrehozott előjel nélküli, egész típusú változó.

`FILE *fopen(const char *allomanynev, const char *mod)`

Az `fopen` függvény megnyitja az adott nevű állományt és visszatér egy adatáramal (állománymutatóval) vagy `NULL` értékkel, ha a megnyitás közben hiba volt. A `mod` kezelési módot megadó paraméter lehetséges értékei:

`"r"`	szöveges állomány megnyitása olvasásra;
`"w"`	szöveges állomány létrehozása írásra, ha az állomány már létezik, akkor előző tartalma elvész;
`"a"`	hozzáfűzés: szöveges állomány megnyitása és írás az állomány végéhez;
`"r+"`	szöveges állomány megnyitása aktuális használatra, azaz olvasásra és írásra;
`"w+"`	szöveges állomány létrehozása aktuális használatra, az állomány korábbi tartalma (ha volt) elvész;
`"a+"`	hozzáfűzés: szöveges állomány megnyitása aktuális használatra és írás az állomány végéhez.

Az aktuális használatnak megfelelő üzemmód megengedi, hogy ugyanazt az állományt olvassuk és írjunk bele. Az írási és olvasási műveletek között az `fflush` függvényt vagy egy állományon belüli pozicionálást végző függvényt kell hívni. Ha a kezelési mód paraméter a megadott betűk után a `b` jelzést tartalmazza (pl. `"rb"` vagy `"w+b"`), akkor ez bináris állományra utal. Az állománynevek hossza max. `FILENAME_MAX` számú karakter lehet. Egy időben legfeljebb `FOPEN_MAX` számú állomány lehet megnyitva.

`FILE *freopen(const char *allomanynev, cost char *mod, FILE *adataram)`

A `freopen` függvény megnyitja az adott nevű állományt jelzett használati móddal és hozzárendeli a megadott adatáramot. A függvény az adatárammal tér vissza vagy ha hiba volt, a `NULL` értékkel. A `freopen` függvényt általában az `stdin`, `stdout` vagy `stderr` adatáramokhoz rendelt állományok megváltoztatására használják.

`int fflush(FILE *adataram)`

Kimeneti adatáramra híva az `fflush` függvény az összes pufferelt, de még ki nem írt adatot kiírja. Bemeneti adatáramra híva a függvényt az eredmény nem definiálható. A függvény normális esetben nulla értékkel, írási hiba esetén `EOF` értékkel tér vissza. Az `fflush(NULL)` kiüríti az összes kimeneti adatáramot.

`int fclose(FILE *adataram)`

A `fclose` függvény az összes pufferelt kimeneti adatot kiírja a megadott adatáramba, törli a még be nem olvasott, de pufferelt adatot, felszabadítja az automatikusan kiosztott pufferterületeket, majd lezárja az adatáramot. A függvény `EOF` értékkel tér vissza, ha bármilyen hiba volt és nulla értékkel máskülönben.

`int remove(const char *allomanynev)`

A `remove` függvény eltávolítja (törli) a megadott nevű állományt, így az állomány megnyitására tett következő kísérlet hibát fog okozni. A függvény nem nulla értékkel tér vissza, ha a hozzáférési kísérlet hibát okozott.

`int rename(const char *reginev, const char *ujnev)`

A `rename` függvény az állomány nevét `reginev`-ről `ujnev`-re változtatja. A függvény nem nulla értékkel tér vissza, ha az állományhoz való hozzáférés sikertelen.

`FILE *tmpfile(void)`

A `tmpfile` függvény `"wb+"` használati móddal létrehoz egy átmeneti állományt, amelyet automatikusan töröl a lezárásakor vagy a program normális lefutásakor. A függvény visszatérésekor normális esetben az adatáramot adja, ill. ha az állomány nem hozható létre, akkor a `NULL` értéket.

`char *tmpnam(char s[L_tmpnam])`

A függvényt `tmpnam(NULL)` formában hívva egy karaktersorozatot generál, ami nem egyezik meg egyetlen létező állomány nevével sem, és visszatéréskor ezt a karaktersorozatot tároló belső statikus tömb mutatóját adja vissza. A `tmpnam(s)` alakú hívás eltárolja a karaktersorozatot az `s`-ben, valamint függvényértékként is visszaadja. Az `s` tömbben legalább `L_tmpnam` számú karakter számára kell hogy hely legyen. A `tmpnam` minden hívásakor egy nevet generál és a program végrehajtása során legalább `TMP_MAX` számú különböző név generálása garantálható. Ügyeljünk arra, hogy a `tmpnam` csak nevet generál és nem állományt.

`int setvbuf(FILE *stream, char *buf, int mod, size_t meret)`

A `setvbuf` függvény az adatáram pufferelését vezérli, és ezt a függvényt kell hívni írás, olvasás vagy bármilyen más állománykezelő művelet előtt. A `mod` paraméter lehetséges értékei:

`_IOFBF` teljes pufferelés;
`_IOLBF` szöveges állományok esetén sorpufferelés;
`_IONBF` nincs pufferelés.

Ha a hívásakor a `buf` paraméter nem `NULL`, akkor a függvény ezt fogja pufferként használni, máskülönben a rendszer rendel pufferterületet a függvényhez. A `meret` paraméter a puffer méretét határozza meg. A függvény hiba esetén nem nulla értékkel tér vissza.

`void setbuf(FILE *adataram, char *buf)`

Ha a függvény hívásakor a `buf` paraméter `NULL`, akkor az adatáram pufferelését kikapcsolja. Minden más esetben a függvény megegyezik a `(void) setvbuf(adataram, buf, _IOFBF, BUFSIZ)` függvénnyel.

B1.2. Formátumozott adatkivitel

A formátumozott adatkiviteli konverziót alapvetően a `printf` függvény különböző változatai végzik.

`int fprintf(FILE *adataram, const char *format, ...)`

Az `fprintf` függvény a `format` karaktersorozatban leírt formátum szerint átalakítja és az adatáramba írja a megadott adatok értékét. A függvény a kiírt karakterek számát adja visszatérési értékként, vagy egy negatív számot, ha hiba volt.

A formátumot leíró karaktersorozat kétféle objektumot tartalmaz: közönséges karaktereket, amelyeket változtatás nélkül bemásol a kimeneti adatáramba, valamint konverziós specifikációkat, amelyek mindegyike az `fprintf` soron következő argumentumának konverzióját és kiíratását vezérli. Az egyes konverziós specifikációk a `%` karakterrel kezdődnek és egy konverziós karakterrel végződnek. A `%` jel és a konverziós karakter között sorrendben a következők helyezkedhetnek el:

– Jelzők (bármilyen sorrendben), amelyek módosítják a specifikációt: mínuszjel, ami a konvertált argumentum balra igazítását írja elő a kiírási mezőben; + jel, ami azt írja elő, hogy a számok kiírása mindig előjellel együtt történjen; szóközkarakter hatására a szám elé szóköz íródik, ha az első karaktere nem előjel; `0` számkonverzió esetén azt írja elő, hogy a kiírási mezőben a szám előtti üres helyek vezető nullákkal töltődjenek fel; # jel a kimeneti formátum megváltoztatását írja elő. `o` esetén a kiírt első számjegy nulla lesz (oktális szám kiírása). `X` vagy `x` esetén a nem nulla szám elé `0x` vagy `0X` íródik (hexadecimális szám kiírása). `e, E, g` és `G` esetén a kiírt szám mindig tartalmazza a tizedespontot és `g` vagy `G` esetén a szám végén lévő értéktelen nullák megmaradnak.

– A kiírási mező minimális szélességét előíró szám: az átalakított argumentum legalább ilyen szélességben (vagy ha szükséges, akkor szélesebb formában) fog kiíródni. Ha az átalakított szám a megadott mezőszélességnél kevesebb karakterből áll, akkor a mező bal széle (ill. ha balra igazítás volt előírva, akkor jobb széle) helykitöltő karakterekkel fog feltöltődni. A helykitöltő karakter normális esetben szóköz, de ha `0`-val való feltöltést írtuk elő, akkor nulla.

– A pont karakter, ami elválasztja a mezőszélességet a pontosságtól.

– A pontosságot meghatározó szám, ami megadja, hogy `e, E` és `f` konverzió esetén a tizedespont után hány számjegyet kell kiírni, vagy `g` és `G` konverzió esetén minimálisan hány számjeggyel íródjon ki egy egész szám (a szükséges szélesség elérése érdekében a szám elé vezető nullák íródnak), vagy a karaktersorozatból hány karaktert kell kiírni.

– Hosszmódosító, ami `h, l` vagy `L` lehet. A `h` azt jelzi, hogy a megfelelő argumentum `short` vagy `unsigned short` formában nyomtatható, az `l` azt, hogy az argumentum `long` vagy `unsigned long` és az `L` pedig azt, hogy az argumentum `long double`.

A mezőszélesség vagy pontosság vagy mindkettő a * jellel is megadható, és ebben az esetben a kívánt érték a következő argumentum(ok)ból, az(ok) konverziójával számítódik ki (az erre a célra használt argumentumoknak `int` típusúaknak kell lenni).

A konverziós karaktereket és jelentésüket a B.1. táblázat tartalmazza. Ha a `%` jel utáni karakter nem konverziós karakter, akkor a függvény viselkedése nincs definiálva.

`int printf(const char *format, ...)`

A `printf` függvény teljesen egyenértékű az `fprintf(stdout, ...)` függvénnyel.

B.1. táblázat. A printf függvény konverziós karakterei

A konverziós karakter	Az argumentum típusa	A nyomtatás módja
d, i	int	decimális szám
o	int	előjel nélküli oktális szám (vezető nullák nélkül)
x, X	int	előjel nélküli hexadecimális szám (a vezető 0x vagy 0X nélkül), a 10...15 jelzése az abcdef vagy ABCDEF karakterekkel
u	int	előjel nélküli decimális szám
c	int	egyetlen karakter
s	char *	karaktersorozatból karaktereket nyomtat a '\0' végjelzésig vagy a pontossággal megadott darabszámig
f	double	[-]m.dddddd alakú decimális szám, ahol d számjegyeinek számát a pontosság adja meg (alapfeltételezés szerint d=6)
e, E	double	[-]m.ddddde xx vagy [-]m.ddddddE xx alakú decimális szám, ahol d számjegyeinek számát a pontosság adja meg (alapfeltételezés szerint d=6)
g, G	double	%e vagy %E alakú kiírást használ, ha a kitevő < -4 vagy >= pontosság, különben a %f alakú kiírást használja. A tizedespont és az utána következő értéktelen nullák nem íródnak ki
p	void *	mutató a géptől függő kiírási formában
n	int *	a printf függvény aktuális hívásakor kiírt karakterek száma beíródik az argumentumba. Az argumentum nem konvertálódik
%		nincs konvertálandó argumentum, egy % jelet ír ki

`int sprintf(char *s, const char *format, ...)`

A sprintf függvény megegyezik a printf függvénnyel, kivéve, hogy a kimenetet az s karaktersorozatba írja, majd a '\0' végjellel lezárja. Az s karaktersorozatnak elegendően hosszúnak kell lennie, hogy az eredményt tárolni tudja. A függvény visszatérésekor a kiírt karakterek számát adja (a végjel nélkül számolva).

`vprintf(const char *format, va_list arg)`
`vfprintf(FILE *adataram, const char *format, va_list arg)`
`vsprintf(char *s, const char *format, va_list arg)`

A vprintf, vfprintf és vsprintf függvények megegyeznek a printf függvénnyel, kivéve, hogy a változó hosszúságú argumentumlistát az arg helyettesíti, ami a va start makróval inicializálható és a va_arg makróval kezelhető. A változó hosszúságú argumentumlistát kezelő eljárások az <stdarg.h> headerben találhatók és használatukat a B7. pontban írjuk le.

B1.3. Formátumozott adatbevitel

A formátumozott adatbeviteli konverziót alapvetően a `scanf` függvény különböző változatai végzik.

`int fscanf(FILE *adataram, const char *format, ...)`

Az `fscanf` függvény a karaktersorozatként megadott formátumleírással vezérelt módon beolvas az adatáramból egy adatot, majd annak konvertált értékét hozzárendeli a soron következő argumentumhoz. *Az adatokat tároló argumentumoknak kötelezően mutató típusúaknak kell lenni!* A függvény akkor fejezi be a működését, ha a teljes formátumleírást feldolgozta. A függvény visszatérési értéke `EOF`, ha bármilyen adatkonverzió előtt állományvége-jelet érzékelt, vagy a beolvasás során hiba történt. Minden más esetben a visszatérési érték a konvertált és argumentumokhoz hozzárendelt bemeneti adatok száma.

A formátumot leíró karaktersorozat konverziós specifikációkat tartalmaz, amelyek közvetlenül felhasználhatók a bemenet értelmezéséhez. A formátumot leíró karaktersorozat tartalma:

- Szóközök vagy tabulátorok, amelyeket a függvény a formátum feldolgozása során figyelmen kívül hagy.
- Közönséges karakterek (nem `%` jel), amelyek várhatóan illeszkednek a bemeneti adatáram következő nem üreshely-karaktereihez.
- Konverziós specifikációk, amelyek a `%` jelből, a `*` opcionális hozzárendelés-elnyomó karakterből, a max. mezőszélességet meghatározó számból (opcionális), a célként megadott argumentum szélességét jelző `h`, `l` vagy `L` karakterből (opcionális), valamint egy konverziós karakterből tevődnek össze.

A konverziós specifikáció a következő bemeneti mező konverziójának módját határozza meg. Normális esetben a konverzió eredménye a megfelelő argumentummal címzett változóba kerül. Ha a `*` hozzárendelés-elnyomó karaktert alkalmaztuk, mint pl. a `%*s` esetben, akkor a függvény a bemeneti mezőt egyszerűen átlépi és nem történik meg a beolvasott érték hozzárendelése a változóhoz. A bemeneti mező nem üres helyekből álló karaktersorozatként van definiálva, és a következő üreshely-karakterig, vagy ha megadtuk, akkor a mezőszélességnek megfelelő számú karakterig tart. Ez azt jelenti, hogy a `scanf` függvény a bemenet keresése közben folyamatosan átolvas a sorhatárokon, mivel az új sor is üreshely-karakternek számít. (Üreshely-karakter a szóköz, a tabulátor, az új sor, a kocsivissza, a soremelés, a függőleges tabulátor és a lapemelés.)

A konverziós karakterek jelzik a bemeneti mező értelmezését. A megengedett konverziós karaktereket a B.2. táblázat tartalmazza.

A `d`, `i`, `n`, `o`, `u` és `x` konverziós karakterek előtt a `h` jelzés állhat, ha a mutatóval címzett argumentum nem `int`, hanem `short` típusú, vagy a `l` jelzés, ha a mutatóval címzett argumentum `long` típusú. Az `e`, `f` és `g` konverziós karakterek előtt állhat az `l`, ha az argumentumként megadott mutató `float` helyett `double` típusú változót címez és az `L`, ha a mutató `long double` típusú változót címez.

`int scanf (const char *format, ...)`

A `scanf(...)` függvény megegyezik az `fscanf(stdin, ...)` függvénnyel.

`int sscanf(char *s, const char *format, ...)`

Az `sscanf(s, ...)` függvény megegyezik a `scanf(...)` függvénnyel, kivéve, hogy a bemeneti karakterek az `s` karaktersorozatból olvasódnak.

B.2. táblázat. A scanf függvény konverziós karakterei

A konverziós karakter	Az argumentum típusa	A beolvasott adat
`d`	`int *`	decimális egész
`i`	`int *`	egész szám, ami lehet oktális (vezető nullákkal) vagy hexadecimális (vezető `0x` vagy `0X` karakterekkel)
`o`	`int *`	oktális egész szám (vezető nullákkal vagy azok nélkül)
`u`	`unsigned int *`	előjel nélküli decimális egész szám
`x`	`int *`	hexadecimális egész szám (a vezető `0x`, ill. `0X` karakterekkel vagy azok nélkül)
`c`	`char *`	karakterek. A következő bemeneti karakterek (alapfeltételezés szerint 1) elhelyezése a kijelölt mezőben. Az üres helyek átlépését (mint normális esetet) elnyomja, ha a következő nem üres karaktert akarjuk beolvastatni, akkor a `%1s` specifikációt kell használni
`s`	`char *`	karaktersorozat (aposztrófok nélkül). A char * mutató egy elegendően nagy karaktersorozatra mutat és a záró `'\0'` jelzést a beolvasás után automatikusan elhelyezi
`[...]`	`char`	a bemeneti karakteráramból beolvassa a zárójelek közötti karakterekkel (illeszkedési halmazzal) megegyező karakterekből álló leghosszabb nem üres karaktersorozatot és lezárja a `'\0'`végjellel. A `[]...]` formában megadott halmaz esetén a `]` karakter a halmaz része lesz. `[^...]` `char *` az illeszkedési halmazzal nem megegyező karakterekből álló karaktersorozat beolvasása és `'\0'` végjellel történő lezárása. A `[^]...]` *formában megadott* halmaz esetén a `]` karakter a halmaz része lesz
`%`		nincs hozzárendelés, a `%` jel, mint karakteres állandó

B1.4. Karakteres adatbevitelt és adatkivitelt kezelő függvények

`int fgetc(FILE *adataram)`
Az `fgetc` függvény visszatér az adatáramból a következő karakterrel. A beolvasott karakter `unsigned char` típusú, amely `int` típusúvá alkul. A visszatérési érték `EOF`, ha az olvasás elérte az állomány végét vagy az olvasás közben hiba történt.

`char *fgets(char *s, int n, FILE *adataram)`
Az `fgets` függvény a következő legfeljebb `n-1` darab karaktert beolvassa az `s` karakteres tömbbe. A beolvasás leáll, ha újsor-karakter fordul elő, de az új sor beíródik az `s` tömbbe. A beolvasott karaktersorozat kiegészül a `'\0'` végjellel. A függvény az `s` tömbbel, ill. hiba vagy állomány vége esetén `NULL` értékkel tér vissza.

`int fputc(int c, FILE *adataram)`
Az `fputc` függvény a `c` karakter `unsigned char` típusra konvertált értékét kiírja az adatáramba. Visszatérési értéke maga a kiírt karakter vagy `EOF`, ha hiba történt.

`int fputs(const char *s, FILE *adataram)`
Az `fputs` kiírja az `s` karaktersorozatot (amelynek nem szükséges `'\n'` újsor-karaktert tartalmazni) az adatáramba és normális esetben nem negatív értékkel, hiba esetén `EOF` értékkel tér vissza.

`int getc(FILE *adataram)`
A `getc` megegyezik az `fgetc` függvénnyel, kivéve, hogy makróként van megvalósítva.

`int getchar(void)`
A `getchar` megegyezik a `getc(stdin)` függvénnyel.

`char *gets(char *s)`
A `gets` függvény beolvassa a következő bemeneti sort az `s` karakteres tömbbe és helyettesíti a sort lezáró újsor-karaktert a `'\0'` végjellel. A függvény az `s` tömbbel vagy állomány vége, ill. hiba esetén `NULL` értékkel tér vissza.

`int putc(int c, FILE *adataram)`
A `putc` megegyezik az `fputc` függvénnyel, kivéve, hogy makróként van megvalósítva.

`int putchar(int c)`
A `putchar` megegyezik a `putc(c, stdout)` függvénnyel.

`int puts(const char *s)`
A `puts` függvény az `stdout`-ra kiírja az `s` karaktersorozatot és lezárja egy újsor-karakterrel. Visszatérési értéke normális esetben nem negatív, hiba esetén `EOF`.

`int ungetc(int c, FILE *adataram)`
Az `ungetch` függvény visszateszi `c` `unsgined char` típusúra konvertált értékét a bemeneti adatáramba, ahonnan az a következő olvasással elővehető. Csak

egy karakter visszahelyezése esetén garantálható a helyes működés. Az EOF karaktert nem lehet visszahelyezni az adatáramba. A függvény visszatérési értéke maga az adatáramba visszahelyezett karakter vagy EOF, ha hiba történt.

B1.5. A közvetlen adatbevitel és adatkivitel függvényei

`size_t fread(void *ptr, size_t meret, size_t nobj, FILE *adataram)`

Az `fread` függvény az adatáramból a `ptr` mutatóval címzett tömbbe olvas legfeljebb `nobj` számú, `meret` méretű objektumot. A függvény visszatérési értéke a beolvasott objektumok száma, ami kisebb lehet a megadott darabszámnál. Az állapot meghatározásához a `feof` és `ferror` makrókat kell használni.

`size_t fwrite(const void *ptr, size_t meret, size_t nobj, FILE *adataram)`

A `fwrite` függvény a `ptr` mutatóval címzett tömbből nobj számú, `meret` méretű objektumot ír ki az adatáramba. A függvény viszatér a kiírt objektumok számával, ami hiba esetén kisebb, mint `nobj`.

B1.6. Állományon belül pozicionáló függvények

`int fseek(FILE *adataram, long offset, int bazis)`

Az `fseek` függvény az adatáram aktuális pozícióját úgy állítja be, hogy a következő olvasás vagy írás ettől a pozíciótól kezdődően fog végbemenni. Bináris állományok esetén az új pozíció a `bazis`-tól számított `offset` számú karaktere lesz, és a `bazis` értéke `SEEK_SET` (az állomány kezdete), `SEEK_CUR` (a régi aktuális pozíció) vagy `SEEK_END` (az állomány vége) lehet. Szöveges állományok esetén `offset` értéke nulla kell legyen vagy az `ftell` függvénnyel előállított értéknek (ilyenkor a `bazis`-nak a `SEEK_SET` értéket kell adni). Az `fseek` hiba esetén nem nulla értékkel tér vissza.

`long ftell(FILE *adataram)`

Az `ftell` függvény visszatér az `adataram`-hoz tartozó állomány aktuális pozíciójával vagy hiba esetén a `-1L` értékkel.

`void rewind(FILE *adataram)`

A `rewind(allomanymutato)` megegyezik az `fseek(allomanymutato, 0L, SEEK_SET); clearerr(allomanymutato)` függvényhívásokkal.

`int fgetpos(FILE *adataram, fpos_t *ptr)`

Az `fgetpos` függvény a `ptr` mutatóval címzett változóba olvassa az `adataram`-hoz tartozó állomány aktuális pozícióját. A kapott érték az `fsetpos` függvényben használható. Az `fpos_t` típus olyan, hogy alkalmas a pozíció tárolására. A függvény hiba esetén nem nulla értékkel tér vissza.

`int fsetpos(FILE *adataram, const fpos_t *ptr)`

Az `fsetpos` függvény az `adataram`-hoz tartozó állományt az `fgetpos` függvénnyel meghatározott és a `ptr` mutatóval címzett helyre eltárolt pozícióba állítja. A függvény hiba esetén nem nulla értékkel tér vissza.

B1.7. Hibakezelő függvények

A könyvtári függvények többsége hiba- vagy állományvége-jelzés esetén beállítja az állapotjelzőket. Ezeket az állapotjelzőket explicit módon lehet beállítani és vizsgálni. Ezenkívül még az `errno` egész típusú kifejezés (amely az `<errno.h>` headerben van deklarálva) tartalmazhat egy hibaszámot, ami további információt szolgáltat a legutoljára előfordult hibáról.

A hibakezelő függvények:

`void clearerr(FILE *adataram)`
: A `clearerr` függvény törli az `adataram`-hoz tartozó, az `EOF`-ot és a hibákat tartalmazó állapotjelzőket.

`int feof(FILE *adataram)`
: A `feof` függvény nem nulla értékkel tér vissza, ha az `adataram`-hoz tartozó `EOF` állapotjelző be van állítva.

`int ferror(FILE *adataram)`
: A `ferror` függvény nem nulla értékkel tér vissza, ha az `adataram`-hoz tartozó hibaállapot-jelző be van állítva.

`void perror(const char *s)`
: A `perror` függvény kiírja az `s` karaktersorozatot, valamint az `errno` hibaszámhoz tartozó, gépi megvalósítástól függően definiált hibaüzenetet. A függvény úgy működik, mintha az

```
fprintf(stderr, "%s: %s\n", s, "hibaüzenet");
```

utasítást adtuk volna ki. A hibaüzenet az `strerror` függvénnyel határozható meg, ennek leírása a B3. pontban található.

B2. Karakteres vizsgálatok: a <ctype.h> header

A `<ctype.h>` headerben vannak a karakteres vizsgálatok függvényei deklarálva. Az egyes függvények argumentuma `int` típusú, amelynek értéke `EOF` vagy `unsigned char` típusban ábrázolható kell hogy legyen. A függvények visszatérési értéke `int` típusú, és nem nulla (logikailag igaz), ha a `c` argumentum kielégíti az adott feltételt, ill. nulla (logikailag hamis), ha nem.

Az egyes függvények (az igaz értékhez tartozó feltételt megadva):

`isalnum(c)`	ha `isalpha(c)` vagy `isdigit(c)` igaz (azaz `c` betű vagy decimális számjegy);
`isalpha(c)`	ha `isupper(c)` vagy `islower(c)` igaz (azaz `c` nagy- vagy kisbetű);
`iscntrl(c)`	ha `c` vezérlőkarakter;
`isdigit(c)`	ha `c` decimális számjegy;
`isgraph(c)`	ha `c` nyomtatható karakter (kivéve a szóközt);
`islower(c)`	ha `c` kisbetű;
`isprint(c)`	ha `c` nyomtatható karakter, beleértve a szóközt is;

`ispunct(c)`	ha `c` nyomtatható karakter, de nem szóköz, betű vagy számjegy;
`isspace(c)`	ha `c` szóköz, lapemelés, új sor, kocsivissza, tabulátor, függőleges tabulátor (üreshely-karakter);
`isupper(c)`	ha `c` nagybetű;
`isxdigit(c)`	ha `c` hexadecimális számjegy.

A hétbites ASCII karakterkészletben a nyomtatható karakterek kódja a `0x20`-tól (`' '`) `0x7E`-ig (`'~'`) terjed. A vezérlőkarakterek kódja `0`-tól (`NUL`) `0x1F`-ig (`US`) terjed és ide tartozik még a `0x7f` (`DEL`) kódja.

Még további két konverziós függvény használható a betűkre:

`int tolower(int c)`	`c` értékét kisbetűvé alakítja;
`int toupper(int c)`	`c` értékét nagybetűvé alakítja.

Ha `c` egy nagybetű, akkor a `tolower(c)` a megfelelő kisbetűvel, a különben magával a `c` értékével tér vissza. Ha `c` egy kisbetű, akkor a `toupper(c)` visszatér a megfelelő nagybetűvel, különben visszaadja a `c` értékét.

B3. Karaktersorozat-kezelő függvények: a <string.h> header

A karaktersorozatot kezelő függvényeknek két csoportja van deklarálva a `<string.h>` headerben. Az első csoportba tartozó függvények neve az `str` karakterekkel kezdődik, a második csoportba tartozóké pedig a `mem` karakterekkel. A `memmove` függvény kivételével a függvények viselkedése definiálatlan, ha átfedő objektumokra alkalmazzuk azokat. Az összehasonlító függvények az argumentumukat `unsigned char` típusú tömbként kezelik.

A következő táblázatban `s` és `t` `char *` típusú, `cs` és `ct` `const char *` típusú, `n` `size_t` típusú, valamint `c` `char` típusra konvertált `int` típusú.

`char *strcpy(s, ct)`
: Az `strcpy` függvény a `ct` karaktersorozatot átmásolja az `s` karaktersorozatba, beleértve a `ct`-t záró `'\0'` végjelet is. A függvény visszatérési értéke `s` mutatója.

`char *strncpy(s, ct, n)`
: Az `strncpy` függvény a `ct`-ből `n` karaktert átmásol `s`-be és visszatér `s` mutatójával. Az `s` végét `'\0'` végjelekkel tölti fel, ha `ct` `n` karakternél rövidebb volt.

`char *strcat(s, ct)`
: Az `strcat` függvény a `ct` karaktersorozatot az `s` karaktersorozat végéhez fűzi (konkatenálja) és visszatér `s` mutatójával.

`char *strncat(s, ct, n)`
: Az `strncat` függvény a `ct` karaktersorozatból `n` karaktert az `s` karaktersorozat végéhez fűz, `s`-t lezárja a `'\0'` végjellel és visszatér `s` mutatójával.

`int strcmp(cs, ct)`
: Az `strcmp` függvény összehasonlítja a `cs` karaktersorozatot a `ct` karaktersorozattal és visszatér negatív értékkel, ha `cs < ct`, nulla értékkel, ha `cs == ct` és pozitív értékkel, ha `cs > ct`.

`int strncmp(cs, ct, n)`
Az `strncmp` függvény összehasonlítja a `cs` karaktersorozat legfeljebb `n` karakterét a `ct` karaktersorozattal és visszatér negatív értékkel, ha `cs < ct`, nulla értékkel, ha `cs == ct` és pozitív értékkel, ha `cs > ct`.

`char *strchr(cs, c)`
Az `strchr` függvény a `c` karakter `cs`-beli első előfordulási helyének mutatójával, ill. ha `c` nem található meg `cs`-ben, akkor `NULL` értékű mutatóval tér vissza.

`char *strrchr(cs, c)`
Az `strrchr` függvény a `c` karakter `cs`-beli utolsó előfordulási helyének mutatójával, ill. ha `c` nem található meg `cs`-ben, akkor `NULL` értékű mutatóval tér vissza.

`size_t strspn(cs, ct)`
Az `strspn` függvény visszatérési értéke a `cs` karaktersorozat elejéről vett és `ct`-ben megtalálható részsorozat hossza. (Az elején egyező rész hossza.)

`size_t strcspn(cs, ct)`
Az `strcspn` függvény visszatérési értéke a `cs` karaktersorozat elejéről vett és `ct`-ben nem megtalálható részsorozat hossza. (Az elején különböző rész hossza.)

`char *strpbrk(cs, ct)`
Az `strpbrk` függvény visszatérési értéke a `cs` karaktersorozat `ct` karaktersorozaton belüli első előfordulásának kezdetét címző mutató, vagy `NULL`, ha `cs` nem található meg `ct`-ben.

`char *strstr(cs, ct)`
Az `strstr` függvény visszatérési értéke a `ct` karaktersorozat `cs`-beli első előfordulásának kezdetét címző mutató, vagy `NULL`, ha a `ct` nem található meg `cs`-ben.

`size_t strlen(cs)`
Az `strlen` függvény visszatérési értéke a `cs` karaktersorozat hossza.

`char *strerror(n)`
Az `strerror` függvény az `n` hibaszámhoz tartozó, a gépi megvalósítástól függő hibaüzenet karaktersorozatának mutatójával tér vissza.

`char *strtok(s, ct)`
Az `strtok` függvény megkeresi az `s` karaktersorozatban a `ct` karaktersorozatból vett karakterekkel határolt tokeneket. (A függvény működésének leírását l. alább.)

Az `strtok(s, ct)` függvény sorozatos hívásával az `s` karaktersorozat tokenekre bontható és az egyes tokeneket a `ct`-ben lévő karakter határolja. A függvény első hívásának `s` nem `NULL` értékével kell történnie, és ekkor a függvény megkeresi az `s`-ben az első, `ct`-ben nem lévő karakterekből felépített tokent, majd az `s` következő karakterét a `'\0'` végjellel felülírva visszatér a tokent címző mutatóval. A további hívásokat az `s` `NULL` értéke jelzi, és a függvény ilyenkor a következő token (amelyet az előzőleg talált token végétől kezd keresni) mutatójával tér vissza. Ha az `strtok` nem talál további tokent, akkor a visszatérési értéke `NULL` lesz. A `ct` karaktersorozat hívásról hívásra változhat.

A `mem` kezdetű függvények különböző objektumokkal mint karakteres tömbökkel való manipulációkra használhatók, és megírásukkal a hatékony adatkezelő eljárások kialakítá-

sa volt a cél. A következő táblázatban s és t void * típusú, cs és ct const void * típusú, n size_t típusú, valamint c unsigned char típusúvá alakított int típusú.

`void *memcpy(s, ct, n)`
A memcpy függvény a ct-ből n karaktert átmásol az s-be és visszatér s mutatójával.

`void *memmove(s, ct, n)`
A memmove függvény megegyezik a memcpy függvénnyel, kivéve, hogy egymást átfedő objektumok esetén is használható.

`int memcmp(cs, ct, n)`
A memcmp függvény összehasonlítja a cs első n karakterét ct-vel. A függvény visszatérési értékei megegyeznek az strcmp visszatérési értékeivel.

`void *memchr(cs, c, n)`
A memchr függvény visszatérési értéke a c karakter cs-beli első előfordulásának helyét címző mutató, vagy NULL, ha c nem található meg cs első n karakterében.

`void *memset(s, c, n)`
A memset függvény elhelyezi a c karaktert az s első n karakterében és visszatérési értéke az s mutatója.

B4. Matematikai függvények: a <math.h> header

A matematikai eljárások függvényei és makrói a <math.h> headerben vannak deklarálva.

Az <errno.h> headerben deklarált EDOM és ERANGE nem nulla értékű egész állandók, amelyek a függvények értelmezési tartomány és értékkészlet hibáját jelzik, a HUGE_VAL értéke pedig pozitív, double típusú szám. Az értelmezési tartomány hiba akkor fordul elő, ha a függvény argumentuma a függvénydefinícióban megadott tartományon kívülre esik. Az értelmezési tartomány hiba esetén az errno beáll EDOM értékére és a visszatérési érték (a hibaüzenet) a gépi megvalósítástól függ. Az értékkészlet hiba akkor fordul elő, ha a függvénnyel kapott eredmény nem ábrázolható double típusú változóval. Ha az eredmény túlcsordult, akkor a függvény visszatérésekor a helyes előjelnek megfelelően állítja be a HUGE_VAL-t és az errno az ERANGE értékének megfelelően áll be. Ha az eredmény alácsordult, akkor a függvény nulla értékkel tér vissza és az errno a gépi megvalósítástól függő módon áll be ERANGE értékére.

A függvények leírását tartalmazó táblázatban x és y double típusú, n értéke int típusú, és midnen függvény visszatérési értéke double típusú. A trigonometrikus függvények argumentumát radiánban kell megadni és az arctg függvény az értékét radiánban adja. A matematikai függvények:

sin(x)	x argumentum szinusza;
cos(x)	az x argumentum koszinusza;
tan(x)	az x argumentum tangense;
asin(x)	az x argumentum árkusz szinusza, az értékkészlet a $[-\pi/2, \pi/2]$ tartomány, $x \in [-1, 1]$;
acos(x)	az x argumentum árkusz koszinusza, az értékkészlet a $[0, \pi]$ tartomány, $x \in [-1, 1]$;
atan(x)	az x argumentum árkusz tangense, az értékkészlet a $[-\pi/2, \pi/2]$ tartomány;

`atan2(y, x)` az y/x érték árkusz tangense, az értékkészlet a $[-\pi, \pi]$ tartomány;
`sinh(x)` az x argumentum szinusz hiperbolikusa;
`cosh(x)` az x argumentum koszinusz hiperbolikusa;
`tanh(x)` az x argumentum tangens hiperbolikusa;
`exp(x)` az e^x exponenciális függvény;
`log(x)` az x argumentum természetes alapú logaritmusa ($\ln(x)$), $x > 0$;
`log10(x)` az x argumentum tízes alapú logaritmusa ($\lg(x)$), $x > 0$;
`pow(x, y)` az x^y alakú hatványfüggvény, értelmezési tartomány hiba lép fel, ha $x = 0$ és $y \leq 0$, vagy ha $x < 0$ és y értéke nem egész szám;
`sqrt(x)` az x argumentum négyzetgyöke, $x \geq 0$;
`ceil(x)` az x argumentumnál nem kisebb legkisebb egész szám, `double` típusra konvertálva;
`floor(x)` az x argumentumnál nem nagyobb legnagyobb egész szám, `double` típusra konvertálva;
`fabs(x)` az x argumentum abszolút értéke ($|x|$);
`ldexp(x, n)` az $x \cdot 2^n$ függvény értéke;
`frexp(x, int *exp)` a függvény az x argumentum érétkét az [1/2, 1) intervallumba eső normált törtrésszé alakítja és ezzel az értékkel tér vissza. A 2 hatványaként értelmezett kitevő a `*exp` című változó ba tárolódik. Ha $x = 0$, akkor az eredmény törtrésze és kitevője egyaránt nulla lesz;
`modf(x, double *ip)` az x argumentum eredeti x előjelével azonos előjelű egész- és törtrészre bontása. Az egészrész az `*ip` című változóban tárolódik, a függvény visszatérési értéke a törtrész lesz;
`fmod(x, y)` az x/y lebegőpontos osztás lebegőpontos maradéka, ami ugyanolyan előjelű mint x. Ha $y = 0$, akor az eredmény a gépi megvalósítástól függ.

B5. Kiegészítő rendszerfüggvények: az <stdlib.h> header

Az `<stdlib.h>` standard headerben vannak deklarálva a számkonverziós, tárkezelő és más hasonló, általános jellegű függvények. Az egyes függvények leírása:

`double atof(const char *s)`
Az `atof` függvény az `s` karaktersorozat tartalmát `double` típusú számmá alakítja. A függvény egyenértékű az `strtod(s, (char**)NULL, 10)` függvénnyel.

`int atoi(const char *s)`
Az `atoi` függvény az `s` karaktersorozat tartalmát `int` típusú számmá alakítja. A függvény egyenértékű az `(int)strtol(s, (char**)NULL, 10)` függvénnyel.

`long atol(const char *s)`
Az `atol` függvény az `s` karaktersorozat tartalmát `long` típusú számmá alakítja. A függvény egyenértékű az `strtol(s, (char**)NULL, 10)` függvénnyel.

`double strtod(const char *s, char **endp)`
Az `strtod` függvény az `s` karaktersorozatot a bevezető üreshely-karakterek elhagyása után egy `double` típusú előtaggá (amely a függvény visszatérési értéke lesz), valamint egy konvertálatlan utótaggá (kivéve, ha `*endp` értéke `NULL`) alakítja. Az utótagot címző mutató a `*endp` helyen tárolódik. Ha az eredmény túlcsordul, akkor a `HUGE_VAL` a helyes előjelnek megfelelő értéket veszi fel; ha az

eredmény alácsordul, akkor a visszatérési érték nulla lesz. Az errno mindkét esetben az ERANGE értékére áll be.

`long strtol(const char *s, char **endp, int alap)`
Az strtol függvény az s karaktersorozatot a bevezető üreshely-karakterek elhagyása után egy long típusú előtaggá (amely a függvény visszatérési értéke lesz), valamint egy konvertálatlan utótaggá (kivéve, ha *endp értéke NULL) alakítja. Ha az alap argumentum értéke 2 és 36 közé esik, akkor a konverzió úgy megy végbe, hogy feltételezi a bemeneti adat adott számrendszerbeli ábrázolását. Ha az alap nulla értékű, akkor az átalakítás oktális, decimális vagy hexadecimális számrendszerbe történik, a bemeneti adat írásmódjától függően (ha a számjegykarakterek sorozata 0 karakterrel kezdődik, akkor oktális, ha pedig 0x vagy 0X karakterekkel, akkor hexadecimális szám lesz az eredmény). A karaktersorozatban szereplő betűk minden esetben a 10 és az (alap - 1) közötti számjegyeket jelölik; a hexadecimális számoknál megengedett a bevezető 0x vagy 0X. Az utótagot címző mutató a *endp helyen tárolódik. Ha az eredmény túlcsordul, akkor a függvény visszatérési értéke az eredmény előjelétől függően LONG_MAX vagy LONG_MIN és az errno az ERANGE értékére áll be.

`unsigned long strtoul(const char *s, char **endp, int alap)`
Az strtoul függvény megegyezik az strtol függvénnyel, kivéve, hogy az eredmény (visszatérési érték) unsigned long típusú és túlcsordulás esetén a visszatérési érték ULONG_MAX értékű.

`int rand(void)`
A rand függvény egy 0 és RAND_MAX közötti pszeudovéletlen egész számmal tér vissza. RAND_MAX értéke legalább 32 767.

`void srand(unsigned int indul)`
Az srand függvény az indul értékével egy új pszeudovéletlen számsorozatot generál. A véletlenszám-generátor kezdeti induló értéke 1.

`void *calloc(size_t nobj, size_t meret)`
A calloc függvény egy nobj számú, egyenként meret méretű objektumot tartalmazó tömb számára lefoglalt tárterület kezdetét címző mutatóval tér vissza. A mutató értéke NULL, ha a helyfoglalási igény nem elégíthető ki. A lefoglalt tárterület nulla értékű bájtokkal van feltöltve.

`void *malloc(size_t meret)`
A malloc függvény egyetlen meret méretű objektum számára lefoglalt tárterület kezdetét címző mutatóval, vagy ha az igény nem elégíthető ki, akkor NULL értékkel tér vissza. A lefoglalt tárterület inicializálatlan.

`void *realloc(void *p, size_t meret)`
A realloc függvény a p mutatóval címzett objektum méretét meret-re változtatja. Az objektum tartalma a régi és új méretek közül a kisebb méretig változatlan marad. Ha az új méret nagyobb a réginél, akkor az új tárterület mutatójával tér vissza, ill. a NULL értékkel, ha az igény nem elégíthető ki (ilyenkor *p változatlan marad).

`void free(void *p)`

A `free` függvény felszabadítja a `calloc`, `malloc` vagy `realloc` függvényekkel lefoglalt, a `p` mutatóval címzett tárterületet. Ha `p` értéke `NULL`, akkor a függvény nem csinál semmit.

`void abort(void)`

Az `abort` függvény a program futásának abnormális befejezését okozza. A függvény működése megegyezik a `raise(SIGABRT)` függvényhívás működésével.

`void exit(int allapot)`

Az `exit` függvény a program futásának normális befejezését okozza. Az `atexit` függvények hívásával kiüríti a megnyitott állományok puffereit, lezárja a megnyitott adatáramokat és a vezérlést visszaadja az operációs rendszernek. Az operációs rendszernek visszaadott allapot állapotjelzés értelmezése a gépi megvalósítástól függ, de a nulla érték mindig a normális (sikeres) befejezést jelenti. Az `allapot` `EXIT_SUCCESS` és `EXIT_FAILURE` értéke szintén használható.

`int atexit(void (*fcn)(void))`

Az `atexit` függvény végrehajtásra előjegyzi az `fcn` függvényt a program normális befejezése esetén. A függvény nem nulla értékkel tér vissza, ha az előjegyzés nem hajtható végre.

`int system(const char *s)`

A `system` függvény végrehajtásra átadja az operációs rendszernek az `s` karaktersorozatot. Ha `s` értéke `NULL`, akkor a függvény nem nulla értékkel tér vissza létező parancsprocesszor esetén. Ha `s` értéke nem `NULL`, akkor a függvény visszatérési értéke a gépi megvalósítástól függ.

`char *getenv(const char *nev)`

A `getenv` függvény visszatérési értéke az operációs rendszertől kapott `nev` nevű karaktersorozat vagy `NULL`, ha nincs karaktersorozat. A függvény működésének részletei a gépi megvalósítástól függenek.

`void *bsearch (const void *kulcs, const void *tabla, size_t n, size_t meret, int (*comp) (const void *kulcsv, const void *datum))`

A `bsearch` függvény a `tabla [0] ... tabla [n - 1]` táblázat elemei (amelyek mérete `meret`) közül megkeresi a kulccsal megegyezőt. A `bsearch` függvény az összehasonlításhoz a `cmp` függvényt használja, amely negatív értékkel tér vissza, ha az első argumentuma kisebb a másodiknál, nullával, ha a két argumentum megegyezik és pozitív értékkel, ha az első argumentum nagyobb a másodiknál. (Az első argumentum a `kulcs`, a második a táblázat megfelelő eleme.) A `tabla` tömb elemeinek növekvő sorrendbe rendezettnek kell lenni. A `bsearch` függvény a `kulcs`-csal megegyező elem mutatójával, ill. ha a `kulcs`-nak megfelelő elem nem található meg a tömbben, akkor `NULL` értékkel tér vissza.

`void qsort (void *tabla, size_t n, size_t meret, int (*cmp) (const void *, const void *)`

A `qsort` függvény növekvő sorrendbe rendezi a `tabla[0]... tabla [n-1]` `meret` méretű objektumokból álló tömböt. Az összehasonlítást a `bsearch` függvénynél alkalmazott `cmp` függvény végzi.

`int abs (int n)`

Az `abs` függvény visszatérési értéke az egész típusú argumentumának abszolút értéke (egész értékként).

`long labs(long n)`

A `labs` függvény `long` típusú vissztérési értéke a `long` típusú argumentum abszolút értéke.

`div_t div (int szaml, int nevez)`

A `div` függvény kiszámítja az egész típusú argumentumokra felírt `szaml/nevez` tört hányadosát és maradékát. Az eredmény egy `div_t` típusú struktúra `quot` és `rem` nevű, `int` típusú tagjaiban tárolódik (a `quot` a hányadost, rem a maradékot tárolja).

`ldiv_t ldiv(long szaml, long nevez)`

Az `ldiv` függvény kiszámítja a `long` típusú argumentumokra felírt `szaml/nevez` tört hányadosát és maradékát. Az eredmény egy `ldiv_t` típusú struktúra `long` típusú, `quot` és `rem` nevű tagjaiban tárolódik (a `quot` a hányadost, `rem` a maradékot tárolja).

B6. Programdiagnosztika: az <assert.h> header

Az assert makró a programdiagnosztika segítésére használható. A makró általános formája:

`void assert(int` *kifejezés*`)`

A makrót az

`assert (`*kifejezés*`)`

formában hívva az az `stderr` állományba az

`Assertion failed:` *kifejezés*`, file` *állománynév*`, line` *nnn*

üzenetet fogja kiírni. Ezután hívható az `abort` függvény, amellyel a program futása befejezhető. Az üzenetben szereplő állománynevet és sorszámot az `assert` az előfeldolgozó rendszerben definiált és kezelt `__FILE__` és `__LINE__` azonosítójú helyekről veszi. Ha az `<assert.h>` header beépítésekor az `NDEBUG` név definiálva van, akkor az `assert` makrót a rendszer figyelmen kívül hagyja (nem hajtja végre).

B7. Változó hosszúságú argumentumlisták kezelése: az <stdarg.h> header

Az `<stdarg.h>` headerben definiált függvények és változók lehetővé teszik ismeretlen számú és típusú argumentumot tartalmazó függvényhívás argumentumlistájának feldolgozását.

Tételezzük fel, hogy a `f(a1, a2, utarg,...)` alakú függvényhívásban `utarg` az utolsó névvel ellátott argumentum, amelyet már a változó argumentumrész követ. Ekkor az `f` függvényhez

```
va_list ap;
```

formában deklarálható egy va_list típusú ap változó (argumentummutató), amely az egyes argumentumokat címzi. Az ap mutatót az első meg nem nevezett argumentumhoz való hozzáférés előtt a va_start makróval a

```
va_start(va_list ap, utarg);
```

módon inicializálni kell. Ezek után a va_arg makró hívásával vehető elő a lista következő, meg nem nevezett argumentuma. A va_arg makró a

```
típus va_arg(va_list ap, típus);
```

formában hívható, és visszatérési értéke az ap által címzett argumentum értéke a megadott típusra konvertálva. A hívás során a va_arg makró az ap mutatót a következő meg nem nevezett argumentumra lépteti. Az f függvényben az argumentumlista feldolgozása után, de még a függvényből való visszatérés előtt egyszer hívni kell a va_end makrót a

```
void va_end(va_list ap);
```

formában, ami lezárja az argumentumlista feldolgozását.

B8. Nem lokális vezérlésátadások: a <setjmp.h> header

A <setjmp.h> header deklarációi lehetőséget nyújtanak a normális függvényhívások és visszatérések elkerülésére, és tipikusan lehetővé teszik, hogy mélyen beágyazott függvényhívásokból közvetlenül térjünk vissza. Az egyes makrók:

int setjmp(jmp_buf env)

A setjmp makró az env változóba elmenti az állapotinformációt. Az elmentett információ a longjmp makró hívásakor használható fel. A setjmp makrót közvetlenül híva a visszatérési értéke nulla. Ha a setjmp-ot a soron következő longjmp makró hívta, akkor a visszatérési értéke nem nulla. A setjmp hívása csak meghatározott programkörnyezetekben fordulhat elő, alapvetően a switch, if utasítások, ill. ciklusok ellenőrző részében és csak egyszerű relációs kifejezésekben. A setjmp használatát a következő programrészlet szemlélteti:

```
if (setjmp(env) == 0)
    /* a setjmp közvetlen hívása */
else
    /* itt következik a longjmp-n keresztüli setjmp
    hívás */
```

void longjmp(jmp_buf env, int ert)

A longjmp makró helyreállítja a korábbi setjmp makró hívásával elmentett, env változóban lévő állapotot, majd a végrehajtás úgy folytatódik, mintha egy setjmp makrót hajtanánk végre és a visszatérés egy nem nulla ert értékkel történik. A longjmp makróban lévő setjmp hívás nincs lezárva. A longjmp hívásakor értéket kapott objektumok értéke hozzáférhető marad, kivéve a setjmp hívásakor érvényben lévő nem volatile automatikus tárolási osztályú változókat, amelyek definiálatlanná válnak, ha az értékük a setjmp hívása után megváltozott.

B9. Jelzések kezelése: a <signal.h> header

A <signal.h> header lehetőséget nyújt a program végrehajtása során előforduló váratlan események, mint pl. külső forrástól érkező megszakításkérés, végrehajtási hiba stb., kezelésére. Az eseményt kezelő függvény általános alakja:

`void (*signal(int sig, void (*handler)(int)))(int)`

A `signal` függvény meghatározza, hogy a rendszer a soron következő jelzést hogyan fogja kezelni. Ha `handler`-nek a `SIG_DFL` lett megadva, akkor a gépi megvalósításban meghatározott alapfeltételezés szerinti kezelést végez, ha pedig a `SIG_IGN` lett megadva, akkor a jelzést figyelmen kívül hagyja. Minden más esetben a `handler`-ben megadott, a jelzés típusának megfelelő argumentummal hívott függvény végzi a jelzés kezelését. A megengedett `sig` jelzések:

`SIGABRT`	program abnormális befejezése, pl. az `abort` függvénytől kapott jelzés;
`SIGFPE`	aritmetikai hiba, pl. nullával való osztás vagy túlcsordulás;
`SIGILL`	illegális függvényhívás, pl. illegális utasítás;
`SIGINT`	interaktív jelzés, pl. megszakításkérés;
`SIGSEGV`	illegális tároló-hozzáférés, pl. címzés a tárhatáron kívülre;
`SIGTERM`	lezárási igény küldése a programhoz.

A `signal` függvény a megadott jelzéshez tartozó `handler` előző értékével vagy hiba esetén a `SIG_ERR` értékkel tér vissza. Amikor a `sig` jelzés bekövetkezik, akkor a `signal` függvény alapállapotba áll vissza, majd `(*hangler)(sig)` formában létrejön a jelzést kezelő függvény hívása. A jelzést kezelő függvényből való visszatérés után a program végrehajtása ott folytatódik, ahol a jelzés megjelenésekor félbeszakadt. A jelzések kezdeti állapota a gépi megvalósítástól függ.

`int raise(int sig)`

A `raise` függvény a `sig` argumentummal megadott jelzést küldi a programnak. Ha a jelzés átadása sikertelen volt, akkor a visszatérési értéke nem nulla.

B10. Dátumot és időt kezelő függvények: a <time.h> header

A <time.h> headerben vannak deklarálva a dátummal és idővel kapcsolatos műveletekhez szükséges adattípusok és függvények. Néhány függvény a helyi időt dolgozza fel, amely különbözhet a naptári időtől, pl. más időzóna miatt. A `clock_t` és `time_t` az idő ábrázolására alkalmas aritmetikai adattípusok, és a `struct tm` a naptári idő komponenseit tartalmazza a következő felosztásban:

`int tm_sec;`	a percet követő másodpercek, 0-tól 61-ig;
`int tm_min;`	az órát követő percek, 0-tól 59-ig;
`int tm_hour;`	az éjféltől eltelt órák száma, 0-tól 23-ig;
`int tm_mday;`	a hónap napja, 1-től 31-ig;
`int tm_mon;`	a január óta eltelt hónapok, 0-tól 11-ig;
`int tm_year;`	az évszám 1900 óta;
`int tm_wday;`	a napok vasárnap óta, 0-tól 6-ig;
`int tm_yday;`	a január 1. óta eltelt napok száma, 0-tól 365-ig;
`int tm_isdst;`	óraátállítás-jelző.

A tm_isdst pozitív, ha az óraátállítás érvényben van, nulla ha nincs érvényben és negatív, ha az erre vonatkozó információ nem áll a rendelkezésünkre.

A dátumot és időt kezelő függvények:

`clock_t clock(void)`
: A clock függvény visszatérési értéke a program kezdete óta eltelt processzoridő, vagy ha ez nem áll rendelkezésünkre, akkor a -1 érték. A processzoridő a clock()/CLOCKS_PER_SEC összefüggéssel számolható át másodpercre.

`time_t time(time_t *tp)`
: A time függvény visszatérési értéke az aktuális naptári idő, vagy -1, ha az nem áll a rendelkezésünkre. Ha tp nem NULL értékű, akkor a visszatérési érték a *tp helyen is eltárolódik.

`double difftime(time_t time2, time_t time1)`
: A difftime függvény visszatérési értéke a time2 - time1 különbség másodpercben kifejezve.

`time_t mktime(struct tm *tp)`
: Az mktime függvény a struktúra *tp helyén lévő helyi időt a time függvénynek megfelelő ábrázolású naptári idővé alakítja. Az idő egyes komponensei a leírtak szerinti tartományba fognak esni. A függvény visszatérési értéke a naptári idő, vagy -1, ha az nem ábrázolható a megfelelő formában.

A következő négy függvény statikus objektumhoz tartozó mutatóval tér vissza és ezek az objektumok más függvényhívásokkal felülírhatók.

`char *asctime(const struct tm *tp)`
: Az asctime függvény a *tp strutúrában található naptári időt a

```
Sun Jan 3 15:14:13 1994\n\0
```

alakú karaktersorozattá alakítja.

`char *ctime(const time_t *tp)`
: A ctime függvény a *tp címen lévő naptári időt helyi idővé alakítja. A függvény egyenértékű az asctime (localtime(tp)) függvényhívással.

`struct tm *gmtime(const time_t *tp)`
: A gmtime függvény a *tp címen lévő naptári időt a koordinált univerzális idővé (UTC) alakítja. A függvény NULL értékkel tér vissza, ha az UTC nem áll rendelkezésre. A gmtime elnevezés történeti okokra vezethető vissza.

`struct tm *localtime(const time_t *tp)`
: A localtime függvény a *tp címen található naptári időt helyi idővé alakítja.

`size_t strftime(char *s, size_t smax, const char *fmt)`
: Az strftime függvény a *tp helyen található dátum- és időadatokat az fmt formátum szerint karaktersorozattá alakítja és elhelyezi az s karaktersorozatban. Az fmt formátumleírás megegyezik a printf függvény formátumleírásával.

A formátumleírásban lévő közönséges karakterek (beleértve a lezáró '\0' karaktert is) átmásolódnak az s karaktersorozatba, a %c alakú elemek pedig a következőkben leírtak szerint, a helyi operációs rendszer megfelelő értékeit felhasználva helyettesítődnek. Az s karaktersorozatban legfeljebb smax számú karakter helyezhető el. Az strftime függvény visszatérési értéke az s-ben elhelyezett karakterek száma (beleértve a lezáró '\0' karaktert is), vagy nulla, ha smax-nál több karaktert kívántunk elhelyezni. A %c alakú formátumspecifikációk:

%a	a nap neve, rövidítve;
%A	a nap neve, teljesen kiírva;
%b	a hónap neve, rövidítve;
%B	a hónap neve, teljesen kiírva;
%c	helyidátum- és helyiidő-ábrázolás;
%d	a hónap napja számmal (01...31);
%H	az óra (24 órás kiírás, 00...23);
%I	az óra (12 órás kiírás, 01...12);
%j	az év napja számmal (001...365);
%m	a hónap számmal (01...12);
%M	a perc (00...59);
%p	az AM vagy PM helyi megfelelője (pl. de vagy du);
%S	a másodperc (00...61);
%U	a hét sorszáma az évben (a hét első napjának a vasárnapot tekintve, 00...53);
%w	a hét napja számmal (vasárnap a 0., 0...6);
%W	a nap sorszáma az évben (a hét első napjának hétfőt tekintve, 000...365);
%x	helyidátum ábrázolás;
%X	helyiidő-ábrázolás;
%y	az évszám, évszázad nélkül (00...99);
%Y	az évszám évszázaddal;
%Z	az időzóna elnevezése, ha van;
%%	% jel.

B11. A gépi megvalósításban definiált határértékek: a <limits.h> és <float.h> headerek

A <limits.h> headerben vannak az egész típusú adatok méreteit megadó állandók definiálva. Az itt megadott értékek a szabvány szerint szóba jöhető minimális nagyságot jelentik, ezeknél nagyobb értékek is használhatók az egyes gépi megvalósításokban.

A következő táblázat a <float.h> headerben definiált, a lebegőpontos aritmetikával kapcsolatos állandók egy részét tartalmazza. A megadott értékek itt is a szabvány szerinti minimumot jelentik, az egyes gépi megvalósítások más, ennél nagyobb értéket is megengedhetnek.

B.3. táblázat. Az egész típusú adatok méretét meghatározó állandók

Azonosító	Érték	Jelentés
CHAR_BIT	8	a bitek min. száma egy `char` típusú változóban
CHAR_MAX	UCHAR_MAX vagy SCHAR_MAX	egy `char` típusú változóban tárolható max. érték
CHAR_MIN	0 vagy SCHAR_MIN	egy `char` típusú változóban tárolható min. érték
INT_MAX	+32767	egy `int` típusú változóban tárolható max. érték
INT_MIN	-32767	egy `int` típusú változóban tárolható min. érték
LONG_MAX	+2147483647	egy `long` típusú változóban tárolható max. érték
LONG_MIN	-2147483647	egy `long` típusú változóban tárolható min. érték
SCHAR_MAX	+127	egy `signed char` típusú változóban tárolható max. érték
SCHAR_MIN	-127	egy `signed char` típusú változóban tárolható min. érték
SHRT_MAX	+32767	egy `short` típusú változóban tárolható max. érték
SHRT_MIN	-32767	egy `short` típusú változóban tárolható min. érték
UCHAR_MAX	255	egy `unsigned char` típusú változóban tárolható max. érték
UINT_MAX	65535	egy `unsigned int` típusú változóban tárolható max. érték
ULONG_MAX	4294967295	egy `unsigned long` típusú változóban tárolható max. érték
USHRT_MAX	65535	egy `unsigned short` típusú változóban tárolható max. érték

B.4. táblázat. A valós típusú adatok méretét meghatározó állandók

Azonosító	Érték	Jelentés
FLT_RADIX	2	a lebegőpontos számábrázolásban a számrendszer alapszáma
FLT_ROUNDS		lebegőpontos kerekítési mód összeadásra
FLT_DIG	6	`float` típusú szám decimális számjegyekben mért pontossága
FLT_EPSILON	1E-5	az a legkisebb `float` típusú *x* szám, amelyre $1.0+x \neq 1.0$
FLT_MANT_DIG		az FLT_RADIX számrendszerű ábrázolásban a mantissza számjegyeinek száma
FLT_MAX	1E+37	egy `float` típusú lebegőpontos szám max. értéke
FLT_MAX_EXP		az a max. *n* szám, amelyre az $FLT_RADIX^{n} - 1$ még ábrázolható
FLT_MIN	1E-37	egy normál `float` típusú lebegőpontos szám min. értéke
FLT+MIN+EXP		az a min. *n* szám, amelyre 10^{n} egy normál számot ad
DBL_DIG	10	`double` típusú szám decimális számjegyekben mért pontossága
DBL_EPSILON		az a legkisebb `double` típusú *n* szám, amelyre $1.0+\neq 1.0$
DBL_MANT_DIG		az FLT_RADIX számrendszerű ábrázolásban a mantissza számjegyeinek száma
DBL_MAX	1E+37	egy `double` típusú lebegőpontos szám max. értéke
DBL_MAX_EXP		az a max. *n* szám, amelyre az $FLT_RADIX^{n}-1$ még ábrázolható
DBL_MIN	1E-37	egy normált `double` típusú lebegőpontos szám min. értéke
DBL_MIN_EXP		az a min. *n* szám, amelyre 10^{n} egy normált számot ad

A C nyelv szabvány bevezetéséből adódó változásai

A könyv első kiadása óta a C nyelv definíciója nagy változásokon ment keresztül. Majdnem minden változás az eredeti nyelv kiterjesztése volt, és a gondos tervezés következtében az új definíció szerinti C nyelv a meglévő gyakorlati alkalmazásokkal kompatíbilis maradt. Az eredeti leírás néhány nem egyértelmű megfogalmazását kijavították és néhány módosítást vezettek be a meglévő gyakorlatnak megfelelően. Sok bejelentett új lehetőség az AT&T rendelkezésre állófordítóprogramjának velejárója volt, és ezt követően adaptálták más C fordítókba. Ezekután az ANSI megfelelő bizottsága ezeket a változásokat meghagyva szabványosította a nyelvet és további jelentős módosításokat is bevezetett. A szabványban foglaltakat előre látva, még annak megjelenése előtt néhány fordítóprogram is megjelent a piacon.

Ez a függelék összefoglalja a könyv első kiadásában és a szabványban definiált C nyelvek magával foglalkozunk, a környezettel (operációs rendszerrel) és a könyvtárral nem. Ez utóbbiak ugyan fontos részét alkotják a szabványnak, de az itteni változások ismertetésének nincs jelentősége, mivel ezekkel a könyv első kiadásában nem foglalkoztunk.

- A Szabvány az előfeldolgozó rendszert sokkal gondosabban definiálta, mint a könyv első kiadása. Az előfeldolgozó rendszer explicit módon a szintaktikai egységekre (tokenekre) bontáson alapszik; új operátorok lettek bevezetve a tokenek láncba fűzésére (`##`) és karaktersorozatok létrehozására (`#`); új vezérlősorok lettek bevezetve (`#elif, #pragma`); explicit módon megengedetté vált a makrók azonos token-sorozattal való újradeklarálása; a karaktersorozat belsejében lévő paramétereket a rendszer a továbbiakban már nem helyettesíti. Az előfeldolgozónak szóló sorok \ jellel való tördelése mindenhol megegendett, nem csak a karaktersorozatokban vagy a makrodefiníciókban. (A részleteket l. az A12. pontban.)

- Az összes belső azonosító nevének szignifikanciája 31 karakterre növekedett; a külső csatolással rendelkező azonosítók nevének szignifikanciájára adott alsó határ megmaradt 6, csak kis- vagy csak nagybetűs karakter. (Számos gépi megvalósítás több karaktert is megenged.)

- A `??` kezdetű trigráf sorozatok bevezetése lehetővé tette a karakterkészletek egy részéből hiányzó karakterek (`#\^[]{}|~`) ábrázolását (l. az A12. pontot). Ügyeljünk rá, hogy a trigráf sorozatok bevezetése változást okozhat a `??` karaktereket tartalmazó karaktersorozatok jelentésében.

- Új kulcsszavakat (`void, const, volatile, signed, enum`) vezettek be, a sikertelen kísérletet jelentő `entry` kulcsszót viszont törölték a szabványból.

- A karaktersorozatokban és karaktersorozat állandókban használható új escape-sorozatokat definiáltak. A \ utáni, a megadott escape-sorozatok között nem szereplő karakterek hatását nem definiálták (l. A2.5.2. pontot).

– Mindenki számára kedvező, triviális változás, hogy a `8` és `9` nem oktális számjegy.

– A Szabvány az állandóktípusának explicit kijelölésére utótagokat vezetett be: az `U` és `L` az egészek, `F` és `L` a lebegőpontos állandók esetén használható. Az utótag nélküli állandók típusára vonatkozó szabályok szintén finomodtak (l. az A2.5. pontot).

– A szomszédos karaktersorozat állandók összefűződnek (konkatenálódnak).

– A Szabvány lehetővé teszi a széles karakterekből álló karakteres és karaktersorozat állandók használatát (l. A2.6.).

– A karakterek, csakúgy mint más egész adattípusok is, explicit módon deklarálhatók előjeles vagy előjel nélküli számábrázolással. Ez a `signed` vagy `unsigned` kulcsszóval valósítható meg. A `double` adattípus szinonímájaként használt `long float` típusjelzést a Szabvány megszüntette, de bevezette a `long double` típust, amellyel extra pontosságú lebegőpontos adatok deklarálhatók.

– Korábban csak az `unsigned char` típus létezett. A Szabvány bevezette a `signed` kulcsszót, amivel a `char` és más egész típusok explicit módon előjelessé tehetők.

– A `void` típus számos gépi megvalósításban évek óta létezett. A Szabvány bevezette a `void*` típust, mint generikus (általános) mutató típust. Ezt a szerepet korábban a `char *` töltötte be. Ugyanakkor explicit szabályokat hoztak a mutatók és az egészek, ill. a különböző típusú mutatók kényszerített típuskonverzió nélküli keveredésének megakadályozására.

– A Szabvány tételesen megadja az aritmetikai adattípusok minimális nagyságát és a megfelelő headerek (`<limits.h>` és `<float.h>`) tartalmazzák az egyes gépi megvalósítások adattípusokra vonatkozó előírásait.

– A felsorolások alkalmazása új a könyv első kiadásához képest.

– A Szabvány a C++ nyelvből átvette a típusminősítő fogalmát, pl. a `const` típusminősítőt (l. az A8.2. pontot).

– A Szabvány szerinti C nyelvben a karaktersorozatok már nem módosíthatók, így csak olvasható tárolóban is elhelyezhetők.

– A könyv első kiadásában alkalmazott „szokásos aritmetikai konverziók" szabálya megváltozott, különösen alapvető a változás, hogy az „egészek esetén mindig az `unsigned` típusnak, lebegőpontos adatok esetén pedig a `double` típusnak van elsősége", valamint a „legkisebb, elegendően nagy típusra való előlépés" szabálymódosult (l. az A6.5. pontot).

– A régi értékadó operátorok, mint pl. az `=+`, eltüntek a Szabványból. Változás még, hogy az értékadó operátorok egyetlen szintaktikai egységet alkotnak, szemben a könyv első kiadásával, ahol üres hellyel elválasztott párt alkottak.

– A fordítóprogramok korábban megengedték a matematikailag asszociatív operátorok számítási szempontból asszociatív kezelését. Ez a szabály a Szabvány bevezetésével megszűnt.

– A Szabvány az unáris – operátor mintájára, szimmetria okokból bevezette az unáris + operátort.

– Egy függvényhez tartozó mutató az explicit * operátor nélkül használható függvénymegnevezésként (l. az A7.3.2. pontot).

– A struktúrák szerepelhetnek értékadásban, átadhatók függvénynek paraméterként és lehetnek függvény visszatérési értékei.

– Az & cím-operátor alkalmazása tömbökre is megengedett és az eredménye a tömböt címző mutató.

– A könyv első kiadásában a `sizeof` operátor `int` típusú eredményt adott, ami számos gépi megvalósításban `unsigned` minősítésű volt. A Szabvány a `sizeof` eredményének típusát gépi megvalósítástól függővé tette, és ehhez bevezetett egy speciális, `size_t` adattípust, ami az `<stddef.h>` standard headerben van definiálva. Hasonló módon kezeli a Szabvány a mutatók különbségének típusát, amelyhez a `ptrdiff_t` adattípus lett bevezetve (l. azA7.4.8. és A7.7. pontokat).

– Az & cím-operátor nem alkalmazható `register` tárolási osztályúnak deklarált objektumokra, még akkor sem, ha a gépi megvalósítás szerint az objektum nem regiszterben tárolódik.

– A léptető kifejezések típusa a bal oldali operandus típusának felel meg, a jobb oldali operandus nem lép elő az eredmény típusának megfelelően (l. az A7.8. pontot).

– A Szabvány megengedi egy tömb utolsó utáni elemét címző mutató létrehozását és annak aritmetikai, ill. relációs kifejezésekben való szerepeltetését (l. az A7.7. pontot).

– A Szabvány – a C++ nyelvből átvéve – bevezette a függvény-prototípus deklarációt, ami a praméterek típusát is tartalmazza, valamint a változó hosszúságú paraméterlista explicit jelölési módját, egyben megadva annak feldolgozási lehetőségét is (l. az A7.3.2., A8.6.3. és B7. pontokat). A régi stílusú függvénydeklaráció továbbra is érvényben maradt, de megszorításokkal.

– Az üres deklarációkat, amelyekben nincs deklarátor és nem deklarálnak legalább egy struktúrát, uniont vagy felsorolást a Szabvány tiltja. Másrészről egy struktúra vagy union címkéjét újra deklaráló deklaráció megengedett, ha az újra deklarálás egy külső érvényességi tartományban történik.

– A valamilyen specifikátort vagy minősítőt nem tartalmazó külső adat-deklarációk (vagyis a puszta deklarátorból álló deklaráció) alkalmazása tilos.

– Néhány gépi megvalósításban a belső blokkban található extern deklaráció érvényességi tartománya kiterjedt a forrásállomány további részére is. A Szabvány világosan rögzíti, hogy az ilyen deklarációk érvényességi tartománya csak az a blokk, amelyben szerepelnek.

– A paraméterek érvényességi tartománya a függvényt alkotó összetett utasítás, így a függvény törzsének tetején lévő változó-deklarációk nem rejthetik el a paramétereket.

– Az azonosítók név-terekbe osztása a könyv első kiadásában szereplő felosztáshoz képest megváltozott. A Szabvány a struktúrák, unionok és felsorolások címkéit egyetlen név-térbe helyezi, és bevezet egy önálló név-teret az utasításcímkék számára (l. az A11.1. pontot). A struktúra- vagy union-tagok nevei annak a struktúrának vagy unionnak a név-teréhez kapcsolódnak, amelynek részét alkotják. (Ez megfelel a régóta fennálló gyakorlatnak.)

– Az unionok inicializálhatók, a kezdetiértéket az union első tagjához rendelik.

– Az automatikus struktúrák, unionok és tömbök inicializálhatók, bár van néhány megkötés.

– Az explicit módon megadott méretű karakteres tömbök pontosan ugyanannyi karaktert tartalmazó karaktersorozat állandóval inicializálhatók (ilyenkor a `'\0'` végjelet egyszerűen kiléteti a rendszer).

– A `switch` utasítás vezérlő kifejezése és `case` címkéje bármilyen egész típusú lehet.

A C PROGRAMOZÁSI NYELV

A Műszaki Könyvkiadó a Szerzők korábban több kiadást megért könyvének egy új, átdolgozott kiadását adja a C nyelvvel ismerkedők kezébe. A sikeres alapkiadás utáni időben az ANSI szabványosította a C nyelvet, ami néhol a nyelv szintaktikájának változtatását vonta maga után. A Szerzők ennek megfelelően dolgozták át (sőt, valójában újraírták) könyvüket, kiegészítve olyan fejezetekkel, mint a szabványosított könyvtári eljárások leírása és használata vagy a UNIX operációs rendszerhez való (most már a szabvány szerint rögzített) kapcsolódás. Külön előnyt jelent, hogy a szabványos C nyelvű programok a legtöbb számítógépen változtatás nélkül futtathatók és a szabványosított C nyelv már rendkívül sok szempontból közelíti a ma gyakran használt C++ nyelvet.

A könyvet mindazoknak ajánljuk, akik meg kívánnak ismerkedni a C nyelvvel (programozók, programozást tanuló egyetemi hallgatók) vagy a korábban megszerzett tudásukat fel kívánják frissíteni. Ezt segíti a könyvben ismertetett számos kidolgozott példa (közöttük néhány ínyencség a programozók számára), valamint a kezdők számára beépített sok megoldandó feladat. Mivel ma a legtöbb programrendszer (operációs rendszerek, fordítóprogramok stb.) C nyelven íródott, így ma már az alapvető számítástechnikai ismeretekhez hozzátartozik a C nyelv ismerete is.

Tartalom:

1. Alapismeretek
2. Típusok, operátorok és kifejezések
3. Vezérlési szerkezetek
4. Függvények és a program szerkezete
5. Mutatók és tömbök
6. Struktúrák
7. Adatbevitel és adatkivitel
8. Kapcsolódás a UNIX operációs rendszerhez
A. függelék: A szabványos C nyelv referencia-kézikönyve
B. függelék: A standard könyvtár
C. függelék: A C nyelv szabvány bevezetéséből adódó változásai

ISBN 963 16 0552 3

9 789631 605525

1580.-